A NEW METHOD FOR THE DESIGN OF FIR DIGITAL FILTERS

by

SCOTT ANTHONY NICHOLS

B.S. Kansas State University, 1986

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Approved by:

[illegible]

Major Professor

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

ACKNOWLEDGMENTS

This author would like to express his appreciation for the support provided in the creation of this manuscript by faculty, friends, and family. Dr. Stephen A. Dyer, my major Professor, is largely responsible for the conception of the method developed herein. I am very grateful for his supervision, patience, and motivation.

A special thanks is also in order for my graduate committee members, Dr. Brian K. Harms and Dr. Christopher M. Sorensen who surely must have suffered a little when perusing the preliminary version. They were able to make some key suggestions for revising the final manuscript.

My parents and sweet "Rosy" also merit thanks for cheering me on when my enthusiasm for writing waned to undetectable levels.

CHAPTER ONE

INTRODUCTION

Introduction to Digital Filtering

Filtering is a process concerned with separating known signals from one another, modifying a signal's characteristics, and attempting to suppress noise or distortion in a desired signal. Examples of filters include cross-overs in stereo speakers, spark plug noise suppressers in automobiles, and equalizers for musical instruments.

While filtering can be accomplished in the time domain, this paper is wholly concerned with filtering in the frequency domain. This means that the filters herein will be discussed in terms of their frequency component selectivity. For example, a low-pass filter would ideally transmit the lower frequency components of a signal unaltered (pass-band) while suppressing the higher frequency components (stop-band). Furthermore, it is usually desired that the transition region, which are the frequencies between those which are fully transmitted and those which are wholly suppressed, be extremely small. This is known in the literature as a brickwall filter due to the abrupt change in amplification between the stop-band and pass-band frequencies. In practice, this brickwall transition cannot be achieved so one measure of a filters effectiveness is how narrow or abrupt this transition region can be made while maintaining smooth characteristics in the stop-band and pass-band.

While filters were originally constructed from discrete components such as resistors, capacitors, and inductors, the advent of modern digital computers has made the digital filter possible. Digital filters perform the same functions as their analog counterparts, but differ in that the filtering operation is done numerically with the characteristics of the filter determined by certain numerical coefficients known as impulse response coefficients. These coefficients completely characterize the filter. The advantages of digital filters over analog are that the filter characteristics can be made arbitrarily close to design values (increase the precision of the coefficients) and it is very easy to change the filter without obtaining different components (change the coefficients). Additionally, since the filter is represented numerically, it can be simulated on a computer. This means that the data to be filtered can be recorded and the filtering leisurely performed at a later date.

Digital filters have two representations, these being infinite impulse response (infinite number of impulse response coefficients) and finite impulse response (finite number of impulse response coefficients). This paper is concerned with the finite impulse response (FIR) digital filter.

Statement of Problem

Various methods for designing FIR digital filters have appeared in the literature within the last two decades. Among these methods are those which strive for optimality in a Chebyshev sense, i.e., possess equal ripple properties, and other simpler representations which seek to directly obtain the filter coefficients. Some well known methods include the Parks-McClellan method [1,2,3], which optimizes the selection of frequencies at which the filter response is specified, and frequency-sampling which obtains the filter by performing an inverse discrete Fourier transform (IDFT) on the desired frequency response. However, these classical methods have their decided disadvantages. It is not trivial to implement and compute a filter via an optimal method. This can discourage the use of these methods in a personal computer oriented environment. On the other hand, one-pass frequency-sampling strategies [4] can be implemented and computed rather quickly but lack specific control over the transition-band frequencies. In addition to that, their response characteristics are often unsuitable at the edges of the transition bands, which tends to prohibit their use in critical filtering applications.

The need exists for a one-pass suboptimal design procedure which will generate a filter quickly, allow exact specification of transition band frequencies, and also attempt to distribute the Chebyshev error at the band-edges throughout more of the filter's spectrum. It will be shown that the discrete cosine transform (DCT) [5] and Lagrange interpolation [6] can be used in conjunction with one another to satisfy these criteria in many cases. In this new method, the DCT is used specifically to recover the approximating-cosine coefficients from which the filter impulse coefficients are derived from. By performing a simple operation on the data prior to transforming, an N/2-point DCT can be implemented in lieu of an N-point IDFT with the attendant decrease in computation effort.

Lagrange interpolation is used to sample an approximation to the filter response at N equally-spaced frequencies. Since the transition band frequencies can be a subset of the guessed extremals, this effectively pins the filter response at the band edges. The other advantage in using Lagrange interpolation is the inherent way in which it assumes band ripple (albeit not equal ripple). This often has a net effect of reducing Chebyshev error close to the transition regions, which the one-pass frequency sampling method tends to accentuate. This is the central theme of the Parks-McClellan approach, which attempts to force the pass and stop-band ripple to increase to an outer limit, thereby distributing the error equally over the entire spectrum of the filter.

The approximation to the FIR filter response can be expressed as a weighted sum of cosine functions. For the case off odd filter length and even symmetry, the form is

$$x(f) = \sum_{k=0}^{n-1} d(k) \cos(2\pi kf), \quad 0.0 \leq f \leq 0.5 \qquad (1)$$

The use of an interpolation strategy based on the guessed set of extremal frequencies is necessary if the DCT is to be used to recover the cosine coefficients. This is due to a lack of correspondence between the arguments of the DCT basis functions and the approximating weighted sum of cosine functions. In other words, the argument of the DCT does not evenly span the frequency range of zero to the Nyquist frequency, as does the approximating function. Hence, an exact interpolation of the function based on the guessed set of extremal frequencies is needed. This will be further delineated as the new procedure is developed.

Thesis Overview

Chapter 2 of this thesis will briefly discuss two well known classical methods for designing FIR digital filters, namely, the one-pass (one iteration of design procedure) frequency-sampling method and the Parks-McClellan method. These methods were chosen due to the differences of their methodology and the availability of literature explaining their implementations. Chapter 3 covers the development of the new method. This includes a discussion of the problems to be overcome in utilizing the DCT for this purpose and also a step-by-step solution of the problem. Chapter 4 describes a design procedure delineating the steps in applying this method in practice. A major part of this thesis is concerned with the software written to design and test the filters. This is entirely covered in Chapter 5. Some mechanical details of the program are presented as well as a detailed example of how to edit and design a filter using the program. Finally, this chapter mentions some quirks that a user should be aware of and some programming hints, should one decide to modify the software. Chapter 6 compares both magnitude responses and design times among the three methods while Chapter 7 summarizes the viability of the new method as a general procedure for designing FIR digital filters.

CHAPTER TWO

REVIEW OF CLASSICAL DESIGN PROCEDURES

Frequency-Sampling Method

A frequency-sampling method for obtaining the filter coefficients is easy to implement and can be computed quickly, but higher-order filters tend to have large deviations at the band-edges. Since band-ripple is not equal throughout the spectrum, it follows that the filter exceeds the design requirements in some cases but does not attain it in others. The frequency-sampling method is based on sampling one period of the desired filter's frequency response. To illustrate, let h(m), $0 \leq m \leq N-1$, be the desired FIR coefficients and H(n), $0 \leq n \leq N-1$, be the DFT coefficients of this sequence. Since any N-point data sequence is completely specified by its N DFT coefficients, the h(n) can be completely recovered from an IDFT of the filter's sampled response. The IDFT of an N-point sequence is defined as

$$h(m) = 1/N \sum_{n=0}^{N-1} H(n)e^{-j2\pi nm/N} \qquad 0 \le n \le N-1 \qquad (2)$$

with the H(n) generally being complex. The transfer function of an FIR filter is represented as

$$H(z) = \sum_{m=0}^{N-1} h(m)z^{-m} \qquad 0 \le m \le N-1 \qquad (3)$$

Substitution of Eqn. (2) into Eqn. (3) yields, after some effort,

$$H(z) = 1/N \sum_{n=0}^{N-1} H(n) \frac{(1 - z^{-N})}{(1 - W^{-n}z^{-1})} \qquad (4)$$

The frequency response can be obtained by the substitution, $z = e^{j\omega T}$. After simplification, this leads to

$$H(\omega') = e^{-j(N-1)\omega T/2} \sum_{n=0}^{N-1} H(n)\ e^{-j(N-1)n\pi/N} \frac{\sin[N(\omega T - 2\pi n/N)/2]}{N \sin[(\omega T - 2\pi n/N)/2]} \qquad (5)$$

where $2\pi n/NT$ is the nth DFT frequency component. This method exactly pins the response at the specified frequency locations but the response between these points is left to the interpolating function, sin(N x)/sin(x) where

$$x = (\omega T - 2\pi n/N)/2.$$

This interpolation can become quite ill-behaved when extremely narrow transition widths are specified. Two additional considerations in the design of FIR filters are: insuring that the impulse coefficients are real and insuring that the filter has linear phase. Real coefficients are obtained by constraining the H(n) such that

$$\tilde{H}(n) = H(N - n) \tag{6}$$

with $1 \leq n \leq N/2 - 1$, for N even and $1 \leq n \leq (N - 1)/2$, for N odd. It is desirable to have real coefficients due to the simplicity of the arithmetic operations and hardware requirements as apposed to performing all operations with complex coefficients. This would unnecessarily slow the filtering process while increasing the requirements for storage registers, etc..

The linear-phase property is of sufficient importance that it too warrants further discussion. Since Fourier analysis indicates that the vast majority of signals encountered in engineering problems can be represented by a weighted sum of sine and cosine basis functions, then it is natural to ask what effect the system has on the phase of each sinusoidal component and if they are all influenced by the system to the same degree. It turns out that in many instances, this is important and it is very desirable to have the property of linear-phase. This means that the time delay of each frequency component is the same and therefore, each component will propagate through the system in the same amount of time. In other words, no distortion due to unequal phase delays will be introduced into the signal. The linear-phase property will achieved by noting that the $H(n) = H e^{j\phi}$ where the H are real constants. In view of this property, the argument ϕ is chosen to be $-(N-1)n\pi/N$, $0 \leq n \leq N-1$; then the product of the H(n) with the $e^{j(N-1)n\pi/N}$ in Eqn. (5) will result in a net phase of zero. This results in a linear-phase property or, in other words, the time delay function will be $(N - 1)/2$, $0 \leq \omega T \leq \pi$.

Parks-McClellan Method

The Parks-McClellan scheme yields the "best" filters that can be obtained in terms of minimizing the overall pass-band and stop-band error. This is achieved when the Chebyshev ripple is equal throughout the frequency spectrum of the filter. However, a general multi-band solution is relatively complicated to implement and the computing effort can be significant for high-order filters. Following is a sketch of the development for FIR linear-phase filters having an odd number of impulse coefficients and even symmetry. The extension to even-length filters or odd symmetries is straight forward and the interested reader can refer to [2] for this development. If h(i) is a casual sequence on the interval $0 \leq i \leq N-1$ then the Z transform of h(i) is defined as

$$H(z) = \sum_{i=0}^{N-1} h(i)\, z^{-i} \tag{7}$$

The Fourier transform of this sequence is

$$H(\omega') = \sum_{i=0}^{N-1} h(i)\, e^{-i\omega n} \tag{8}$$

with ω' defined hereafter as $\omega' = e^{j\omega}$ for convenience. An odd-length, even-symmetry, linear-phase filter can be described as follows

$$H(\omega') = G(\omega')\, e^{j(N-1)/2} \tag{9}$$

where

$$G(\omega') = \sum_{i=0}^{n-1} d(i) \cos(\omega i) \tag{10}$$

and $n = (N-1)/2 + 1$, $d(0) = h(n-1)$ and $d(i) = 2h(n-i-1)$ for $i=0,1,\ldots,n-1$. The formulation of the problem is to find the d(i) such that the function $G(\omega')$ is the mini-max approximation to the desired function $D(\omega')$. In pursuit of this goal, we define an error function

$$E(\omega') = W(\omega')\,[\,D(\omega') - G(\omega')\,] \qquad (11)$$
$$= (-1) \;\; x \;\; \rho$$

with $W(\omega')$ defined as a ratio of the Chebyshev error in any band compared to the normalized ripple of an arbitrarily chosen band. At least one or more bands of the filter must have an error weighting function $W(\omega')$ of unity to give meaningful deviation ratios.

Eqn. (11) now defines a problem in Chebyshev approximation whereby the d(i) are found such that the error function $E(\omega')$ is minimized over the (possibly disjoint) intervals of interest. A well-known theorem which is key in the solution of Chebyshev approximation over disjoint intervals is the Alternation Theorem [2]. If some function $X(\omega')$ is a linear combination of r cosine functions, then a necessary and sufficient condition for $X(\omega')$ to be the best unique weighted Chebyshev approximation to $D(\omega')$ over subsets of $[0, \pi]$ is that $E(\omega')$ contains at least r+1 extremal frequencies, $\omega'(i)$, $i=0,1,\ldots,r$. In other words,

$$E[\omega'(i)] = - E[\omega'(i+1)] \text{ for } i=0,1,\ldots,n \quad [2].$$

The solution of the nonlinear system of equations defined by Eqn. (11) can be obtained from a variety of methods, the most efficient being the Remez exchange [1,2,3] which is the method chosen by Parks and McClellan. The gist of this procedure is as follows.

1) Choose n+1 frequencies. These are estimates of the extremals of the approximation to the desired function and usually are equally spaced.

2) Calculate an estimate of ρ.

3) Use Lagrange interpolation over n points to obtain an approximation to the desired function $D(\omega')$ where n is defined as $n = (N-1)/2 + 1$.

4) Compute the error function defined in Eqn. (11).

5) Search this error function for the n+1 (or more) frequencies at which the error function exceeds the calculated value for the deviation.

4) If more than n+1 extremals exist, discard those which exhibit the least amount of deviation or those which do not satisfy the Alternation Theorem. With respect to the endpoints of the error function, if N+2 extremals exist, then retain the extremals yielding the largest absolute value of deviation.

5) If no extremals changed in location from the previous iteration, then the approximation problem is finished; otherwise, repeat the process starting at step 2,

utilizing the new set of extremals.

6) Obtain the d(i) in Eqn. (10) via an IDFT.

7) Obtain the impulse coefficients in Eqn. (8) from the following relations (exemplified for odd length, symmetrical filters).

   $d(0) = h(n-1)$, $d(i) = 2h(n-i-1)$ for $i=0,1,\ldots,n-1$.

CHAPTER THREE

DEVELOPMENT OF THE NEW METHOD

A new method will now be presented which overcomes some of the disadvantages of the classical methods. The goal at hand is that of obtaining a (suboptimal) FIR filter whose frequency response will be approximated by a series of the form in Eqn. (12). The coefficients d(k) are to be found.

$$x(f) = \sum_{k=0}^{n-1} d(k)\cos(2\pi kf), \qquad 0 \leq f \leq 0.5 \qquad (12)$$

It is observed that the DCT is one of a class of discrete, weighted Chebyshev polynomials which possesses an inverse transform with a form similar to that in Eqn. (12). The DCT is given by

$$L(k) = \sqrt{2/n}\sum_{m=0}^{n-1} x(m)\cos([2m+1]k\pi/2n) \qquad k=1,\ldots,n-1 \qquad (13)$$

$$L(0) = \sqrt{1/n} \sum_{m=0}^{n-1} x(m)$$

the inverse DCT (IDCT) is

$$x(m) = L(0)/\sqrt{n} + \sqrt{2/n} \sum_{k=1}^{n-1} L(k) \cos([2m+1]k\pi/2n) \qquad m=0,1,\ldots,n-1 \qquad (14)$$

We note that Eqns. (12) and (14) differ by the starting index, an additional sum term of $L(0) / \sqrt{n}$, and a multiplication factor of $\sqrt{2/n}$. A modification of the original data sequence x(m), m = 0,1,...,n-1, is needed to alleviate these differences. The DCT of the transformed x(m) will then yield the d(k) in (12) exactly.

We first deal with the difference in indexing between (12) and (14) by rewriting (14) as

$$x(m) = \sqrt{2/n} \sum_{k=0}^{n-1} L(k) \cos([2m+1]k\pi/2n) - L(0)(\sqrt{2} - 1) / \sqrt{n} \qquad m=0,1,\ldots,n-1 \qquad (15)$$

We next eliminate the factors of $\sqrt{2/n}$ and $L(0)(\sqrt{2} - 1)/ \sqrt{n}$. To accomplish this, assume a new data sequence

$$x'(m) = a(m)[x(m) + y(m)], \qquad m=0,1,\ldots,n-1$$

then

$$L(0) = \sqrt{1/n} \sum_{m=0}^{n-1} a(m) [x(m) + y(m)]$$

and

$$L(k) = \sum_{m=0}^{n-1} a(m)[x(m) + y(m)] \cos([2m+1]k\pi/2n), \qquad k=1,2,...,n-1 \tag{16}$$

It also follows that the inverse transform can be written as

$$a(m)[x(m) + y(m)] = \sum_{m=0}^{N-1} L(k) \cos([2m+1]k\pi/2n) - L(0)(\sqrt{2} - 1) / \sqrt{n} \qquad m=0,1,...,n-1 \tag{17}$$

Letting $a(m) = \sqrt{2/n}$, we can rewrite (17) as

$$x(m) = \sum_{k=0}^{n-1} L(k) \cos([2m+1]k\pi/2n) - y(m) - L(0)(\sqrt{2} - 1)/\sqrt{2} \qquad m=0,1,...,n-1 \tag{18}$$

If we force $y(m) + L(0)(\sqrt{2} - 1)/\sqrt{2} = 0$ for $m=0,1,...,n-1$, then we have accomplished our goal. To attain this, we note that the original definition for the L(0)-term in the DCT is

$$L(0) = \sqrt{1/n} \sum_{i=0}^{n-1} a(m)[x(m) + y(m)].$$

Hence,

$$L(0) = \sqrt{1/n} \sum_{i=0}^{n-1} \sqrt{2/n}\, x(m) + \sqrt{1/n} \sum_{i=0}^{n-1} \sqrt{2/n}\, y(m)$$

Substituting for L(0) now yields

$$y(m) + \left[\sqrt{1/n}\sum_{i=0}^{n-1}\sqrt{2/n}\; x(i) + \sqrt{1/n}\sum_{i=0}^{n-1}\sqrt{2/n}\; y(i)\right](\sqrt{2} - 1)/\sqrt{2} = 0$$
$$m=0,1,\ldots,n-1 \qquad (19)$$

After simplifying, we get

$$y(m) + (\sqrt{2} - 1)/n \sum_{i=0}^{n-1} y(i) + (\sqrt{2} - 1)/n \sum_{i=0}^{n-1} x(i) = 0$$
$$m=0,1,\ldots,n-1 \qquad (20)$$

From this equation, it is clear that y(m) = y is a constant for all m=0,1,...,n-1. Eqn. (20) then can be solved for y in terms of the x(i) as follows.

$$(\sqrt{2} - 1)/n \sum_{i=0}^{n-1} y(i) = ny(\sqrt{2} - 1)/n$$
$$= (\sqrt{2} - 1)y$$

Hence,

$$y + (\sqrt{2} - 1)y + (\sqrt{2} - 1)/n \sum_{i=0}^{n-1} x(i) = 0$$

Finally, we arrive at a closed-form expression for y in terms of the x(i) as

$$y = (1 - \sqrt{2})/(n\sqrt{2}) \sum_{i=0}^{n-1} x(i) \qquad (21)$$

This yields the modification for the original data sequence of x(m), m=0,1,...,n-1 as

$$x'(m) = \sqrt{2/n}\,[x(m) + y] \qquad m=0,1,\ldots,n-1 \qquad (22)$$

where y is given in Eqn. (21). Performing a DCT on this new data sequence will yield the d's in Eqn. (12).

In the frequency-sampling method, the N impulse coefficients are obtained by performing an N-point IDFT of the desired frequency response. This does not permit the transition regions to be specified exactly since the transitions can be specified only at discrete frequency intervals. Additionally, some procedure must be employed to guess at the response in the transition region. The frequency-sampling example herein employs a simple linear interpolation scheme.

As an alternative using the DCT, consider, for example, an even-symmetry filter with an odd number N of impulse coefficients. In this situation, the number n of approximating cosines is n = (N - 1)/2 + 1, where

$$x(f) = \sum_{k=0}^{n-1} d(k) \cos(2\pi kf), \qquad 0 \leq f \leq 0.5 \qquad (1,23)$$

Comparing this with

$$x(m) = \sum_{k=0}^{n-1} L(k) \cos([2m+1]k\pi/2n) \qquad m=0,1,\ldots,n-1 \qquad (24)$$

we notice that the arguments of the cosines have the correspondence

$$2\pi kf \longleftrightarrow (2m+1)2k\pi/4n \tag{25}$$

Hence, the frequency components of the DCT are

$$f = (2m + 1)/4n \qquad m=0,1,\ldots,n-1 \tag{26}$$

The L(k) are obtained, using the DCT, as

$$L(k) = \sqrt{2/n} \sum_{m=0}^{n-1} x'(m) \cos([2m+1]k\pi/2n) \qquad m=0,1,\ldots,n-1$$

The final problem is to obtain values for x'(m) at the frequencies f = (2m + 1)/4n for m=0,1,...,n-1. This can be done via the Lagrange interpolation formula.

$$x'(f) = \frac{\sum_{k=0}^{n-1} a_k c_k/(x_f - x_k)}{\sum_{k=0}^{n-1} a_k/(x_f - x_k)} \tag{27}$$

where $x = \cos(2\pi f)$, and $x_k = \cos(2\pi f_k)$. We note that the f's are the frequencies derived from Eqn. (26), and the f_k's are the original estimates of the n+1 extremals used to generate the deviation

$$\rho = \frac{\sum_{i=0}^{n+1} a_i D(f_i)}{\sum_{i=0}^{n+1} \frac{a_i(-1)^i}{W(f_i)}} \tag{28}$$

where D(f) is the desired response, W(f) is the weighting function, and the a_i's are given by

$$a_i = (-1)^i \sum_{j=0, j\neq i}^{n+1} 1/(x_j - x_i) \tag{29}$$

We notice that, in this procedure, the edges of the transition band may be specified exactly, the weighting function can be arbitrary, and any of the four cases of filters (symmetric/antisymmetric; even/odd) may be obtained.

CHAPTER FOUR

DESIGN PROCEDURE

The following describes a design procedure for obtaining FIR digital filters via the new method.

1) Compute n+1 equally-spaced frequency values contained in the intervals

   $$f \in [0.0, f_1] \cup [f_2, f_3] \ \ldots \ \cup [f_j, 0.5]$$

   where f_j represents the beginning transition frequency of the last filter band. These are the frequencies used in the equation $x_k = \cos(2\pi f_k)$.

2) Obtain n equally-spaced frequencies by indexing through Eqn. (26). These are the frequencies used in equation $x = \cos(2\pi f)$.

3) Sample an approximation of the filter's magnitude response at these frequencies via Lagrange interpolation, Eqn (27).

4) Compute the constant

$$y = (1 - \sqrt{2})/(n \sqrt{2}) \sum_{i=0}^{n-1} x(i)$$

5) Compute the modified frequency response

$$x'(m) = \sqrt{2/n}\ [x(m) + y], \quad m = 0,1,\ldots,n-1$$

6) Perform an N/2-point DCT on the new sequence to obtain the coefficients to the approximating cosines.

7) Obtain the filter impulse coefficients from these cosine coefficients by the following relation.

$h(n-1) = d(0), \quad 2h(n-i-1) = d(i)$ for $i=0,1,\ldots,n-1$.

CHAPTER FIVE

PROGRAM

Program ROLF.EXE

A program was developed in order to expedite the process of developing and simulating the filters and also for the purpose of comparing relative performance between design procedures. Care has been taken to modularize the design while keeping an appropriate level of visibility between modules in anticipation of the needs of future users. The overall architecture of the program is functionally equivalent to an a Hewlett-Packard reverse-Polish-notation calculator with the scalar stack registers being replaced by structures of complex vectors and a variety of extensions being added. The extensions include filtering and general signal processing. While the program provides the shell for many different functions, only the filtering, stack manipulations, and plotting routines will be described in detail since they are germane to the results of the research completed herein. A flowchart of the program is provided in Figure 1, which illustrates the constituent modules relating to the FIR filter design section.

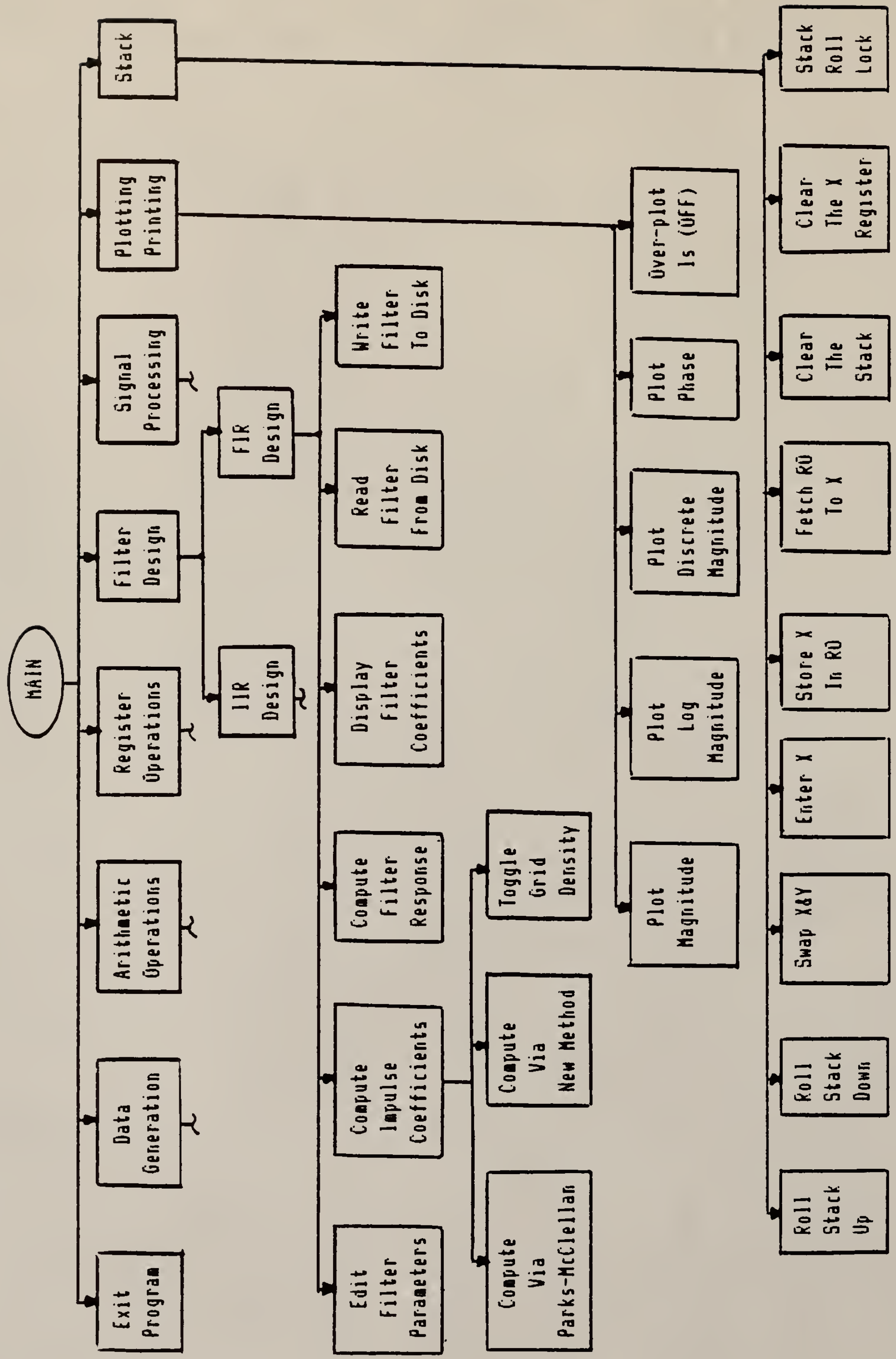

Figure 1. Block layout of program.

Upon entry into the program the primary menu is displayed to the user.

```
-------------------------------------------------------------
| Register   | Length  |  Mode   | Type  |    Contents       |
------------------------------------------------------------|
|    R0      |      0  |         |  REAL |  empty            |
|------------|-----------------------------------------------|
|    T       |      0  |         |  REAL |  empty            |
|    Z       |      0  |         |  REAL |  empty            |
|    Y       |      0  |         |  REAL |  empty            |
|    X       |      0  |         |  REAL |  empty            |
-------------------------------------------------------------

1:  Data generation
2:  Arithmetic operations
3:  Register operations
4:  Filter design
5:  Signal processing
6:  Register input/output
7:  Load second copy of COMMAND.COM
    <RETURN> Stack

>>

=============================================================
```

The stack-register monitor (where applicable) is for the convenience of the user in tracking the contents and attributes of the various registers. These registers are the structures containing the complex arrays in which are stored the filter coefficients, filter response, and all general data. Each register contains the information displayed in the primary menu, i.e., the complex vector, register length, mode (polar or rectangular), type (real or complex), and a message indicating the last action or contents of that register. The filter section of the program is invoked by choosing menu selection (4). The following menu is then displayed:

```
============================================================

0:  Exit
1:  FIR
2:  IIR

>>

============================================================
```

Selecting Option (1) takes us to the finite-impulse response digital filter design section, with which this paper is concerned. Upon selection of menu option (1), the following display is presented:

```
============================================================

0:  Exit
1:  Edit filter parameters
2:  Compute impulse coefficients
3:  Compute filter response
4:  Display filter coefficients
5:  Read filter from disk
6:  Write filter to disk
    <RETURN> Stack

>>

============================================================
```

It should be noted at this time that the parameters which describe the filter to be designed are not resident in the stack registers but rather are placed in a special FIR filter structure. This means that reading a filter from the disk or editing a filter in memory will not affect the stack. The different menu options will now be reviewed.

Editing/Selecting Filter Prototypes

This allows the parameters constraining the design characteristics of the FIR filter to be changed by the user. In particular, the response type, symmetry, length of impulse response, band value, and band weighting must be specified. Following is an example run through this editing procedure. The user is first presented with the following prompt:

```
==============================================================

1:  Brickwall
2:  General

>>
==============================================================
```

Choosing a brickwall response is appropriate for multibanded filters such as low-pass or band-pass. The general response type will obtain a Chebyshev approximation to a real data sequence in the X register such as 1/f, etc.. The other utilities of the program can eventually be used to create such a response. The next choice to be made is:

```
============================================================

1:   Symmetrical
2:   Asymmetrical

>>

============================================================
```

which describe the symmetry of the impulse coefficients of the resultant filter. Following this menu is a request for the number of impulse coefficients:

```
============================================================

Filter Length

>>

============================================================
```

and the number of distinct bands of the filter:

```
=============================================================

Number of distinct bands

>>

=============================================================
```

In specifying the number of bands, for example, a general response type would possess one distinct band, a low-pass filter would possess two distinct bands, a band-pass three, and so on.

The following two menus will appear only if the user specifies a brick-wall filter type. In this case, the user is requested to specify the magnitude and band weight for each distinct band:

```
=============================================================

Enter magnitude      for band[i] >
Enter band weighting for band[i] >

=============================================================
```

The magnitude will normally be 1.0 for pass bands and 0.0 for stop bands, although filters with arbitrary characteristics can be produced by specifying, say 0.8 for one of the pass bands. The band weighting is a ratio of the maximum allowable error in the band under question to that of another band with an error weighting of unity. At least one band of the filter must have a weighting of 1.0 for the ratios to yield meaningful results. For example, if one wanted the stop-band ripple of a low-pass filter to be twice that of the pass-band ripple, then a weighting of 2.0 could be assigned to the stop-band ripple and 1.0 to the pass-band ripple. The final parameters to be specified are the transition frequencies (still for brick-wall only). Also note the difference in menus for the first, last, and middle bands of the filter. This is due to the fact that the first transition frequency is always zero while the last corresponds to the Nyquist frequency. Hence, the user is not requested to input their values. For the first distinct band:

```
============================================================

Enter upper transition for band [i] >

============================================================
```

For any middle bands that may exist:

```
================================================================

Enter lower transition for band [i] >
Enter upper transition for band [i] >

================================================================
```

For the last distinct band:

```
================================================================

Enter lower transition for band [i] >

================================================================
```

The values to be specified here are a ratio of the desired frequency divided by the sampling frequency, i.e., f/fs.

Computing Impulse Coefficients

This menu selection displays the following sub-menu selection:

```
================================================================

0:  Exit
1:  Compute via Parks-McClellan
2:  Compute via New Method
3:  Toggle frequency grid density: (16)
    <RETURN> Stack

>>

================================================================
```

This is the section where the actual filter coefficients are computed. They are returned in the X register with the type attribute set to REAL and the mode undefined. Note that in order to compute the coefficients, a filter must have previously been edited or read from the disk, or in the case of a generalized filter response, the desired prototype must be present in the X register.

Parks-McClellan:

Computes the filter coefficients by invoking the Remez Exchange algorithm.

New Method:

Computes the filter coefficients by using Lagrange interpolation and the DCT.

Adjusting The Dense Frequency Grid:

This value toggles over the following set of values:

$$\text{density} \in [16, 20, 25, 5]$$

These values control the density of the frequency grid [x] over which the error function is evaluated in the Parks-McClellan routine, i.e., frequency grid [0 , {(N-1)/2 + 1} * density]. The closeness of the Chebyshev approximation and filter design time of both the new method and Parks-McClellan method is affected by this value. Smaller values significantly reduce design times but yield lower quality approximations. The default value of (16) provides good overall performance. A value of (5) yields extremely fast design times for large filters while often suffering only slight degradation in filter quality. Large values are sometimes necessary when extremely narrow transition regions are desired. The strategy is to try the filter and if the procedure aborts prematurely, then the grid density can be increased.

Computing Filter response

Valid filter coefficients must be present in the X register prior to invocation of this routine. Functionally, it zero-pads the coefficients out to the next power of two and performs an FFT on the resulting sequence.

Displaying Filter Coefficients

This allows the impulse coefficients to be printed to the screen. For long filters, the user may abort the display prior to completion with no ill effects. Note that the X register needs to contain the impulse coefficients and not the filter response or some other extraneous data.

Reading/Writing Filter Prototypes To Disk

This allows a previously edited filter to be read into the

FIR filter structure at which point the filter can be edited or the coefficients computed. An edited filter can also be saved to the disk with any user specified name. If the file name previously exists, then the user is prompted for permission to overwrite it.

Stack/Register Manipulations

This utility allows the stack register contents to be manipulated from nearly anywhere in the program. This eliminates the frustrating and time consuming return from deep within a complex hierarchy to invoke stack control from the main menu. To invoke it (where it is available), just hit <RETURN>. Similarly, <RETURN> will exit the stack functions, returning control back to the previous context. From the menu, it is clear that a variety of utilities exist to save, swap, push, pop, and roll the stack registers. The stack menu appears as follows:

```
============================================================

0:  Exit

1:  Roll stack up
2:  Roll stack down
3:  Swap X and Y
4:  Enter (duplicate) X
5:  Store X in RO
6:  Fetch RO to X
7:  Clear the X register
8:  Clear the stack
9:  Toggle stack roll lock: (UNLOCKED)
    <RETURN> Exit
>>

============================================================
```

Plotting Impulse Coefficients and Response

This utility is invoked from the main menu (not from the filter design context) and provides graphic display of the impulse coefficients, filter response (magnitude, log magnitude, and discrete), and phase in radians. Additionally, the data can be printed to the screen in numeric format. Although the X register normally provides the data, the "over plot" select can be toggled "ON" to plot both the X and Y registers on the same screen. Obviously, both registers need to represent the same kinds of entities to provide meaningful results.

Things To Watch For

A couple of factors can cause an abortion of the design procedures prior to obtainment of a filter. In the worst case, the user is returned to the operating system prompt (gasp) due to a divide-by-zero. Due to the logistics in computing the Remez exchange, preventive logic seemed like a bad idea. Typical causes are as follows:

* Specifying too narrow of a transition region such that both cutoff frequencies fall within one increment of the dense grid of cosines.
* Specifying too wide of a transition region such that the estimate of the error is extremely small and round-off error begins to dominate in the Lagrange interpolation routine.
* Specifying the filter order so large that round-off error dominates in the Lagrange interpolation.

Solutions to these problems are simply to adjust the constraints on the filter to a more reasonable level.

<u>Hints to Programmers</u>

The two main data structures in the program are the stack registers and the FIR filter structure. These will be discussed briefly in turn. The stack registers are defined in stackops.c using the structure template defined in rolf.h.

```
typedef struct
   {
   int len,          /* register length                     */
       mode,         /* RECT or POLAR  format               */
       type,         /* REAL or COMPLEX data types          */
       contents;     /* array index pointing to string      */
                     /* description of contents or action */
       COMPLEX *reg; /* actual data storage                 */

    } REG x, y, z, t, etc...;
```

These objects are declared as statics to maintain the least amount of visibility while reducing the possibility of definition conflicts. If, for example, one wants to add a utility tc the program, the following template exemplifies what preparation needs to be done to gain access to one of the registers, in this case, taking an FFT of the X register.

```
void my_function()
{
    REG *my_reg;         /* define  a pointer to any reg  */

    my_reg = get_reg(X); /* returns a pointer to X reg    */
                         /* X,Y,Z,T, and RO are available */
    my_reg->len = 1024;

    fft(my_reg, my_reg->len, FORWARD_TRANSFORM);

    my_reg->mode = RECT;    /* RECT is defined in smath.h */
    my_reg->type = CMPLX;   /* CMPLX defined in smath.h   */
    return;
}
```

The data object which specifies the FIR filter parameters or specifications uses the following structure templates:

```
typedef struct
    {
    double lw,      /* lower cut-off frequency */
           up;      /* upper cut-off frequency */
    } FREQUENCIES;
```

and

```
typedef struct
    {
    int type,       /* even or odd symmetry            */
        response,   /* brickwall or general            */
        order,      /* number of impulse coefficients */
        nobands;    /* no of distinct filter bands     */

    double band_value[NOBANDS],  /* magnitude response */
           band_weight[NOBANDS]; /* weighting ratio    */

    FREQUENCIES tran_freq[NOBANDS];
    } FIR_SPECS;
```

There is currently only one storage object with which to store the specifications of a filter. This object has no interaction with the stack registers, i.e., it can be edited with the stack registers being unaffected.

CHAPTER SIX

EXAMPLES AND RESULTS

Comparing Filter Magnitude Responses

Four examples are presented so that relative performance, in terms of response characteristics and design times, can be illustrated and compared. The first example has an impulse length of 21 while the other examples were chosen to have 95 impulse-response coefficients. All examples represent filters with even symmetry. In every example, the response of the new method is represented by a solid line while the classical design procedures are represented by a dashed line.

Figures 2 and 3 represent LP filters of 21 impulse coefficients defined over the subintervals $f \in [0,.2] \cup [.25,.5]$. Figure 2 compares the frequency-sampling method with the new method. The new method provides about 6 dB additional attenuation in the stop-band while the frequency-sampling method provides a smoother pass-band. Figure 3 compares the new method with the Parks-McClellan method. As is usually the case, the first stop-band lobe from the new method exceeds that which is obtained from the Parks-McClellan method.

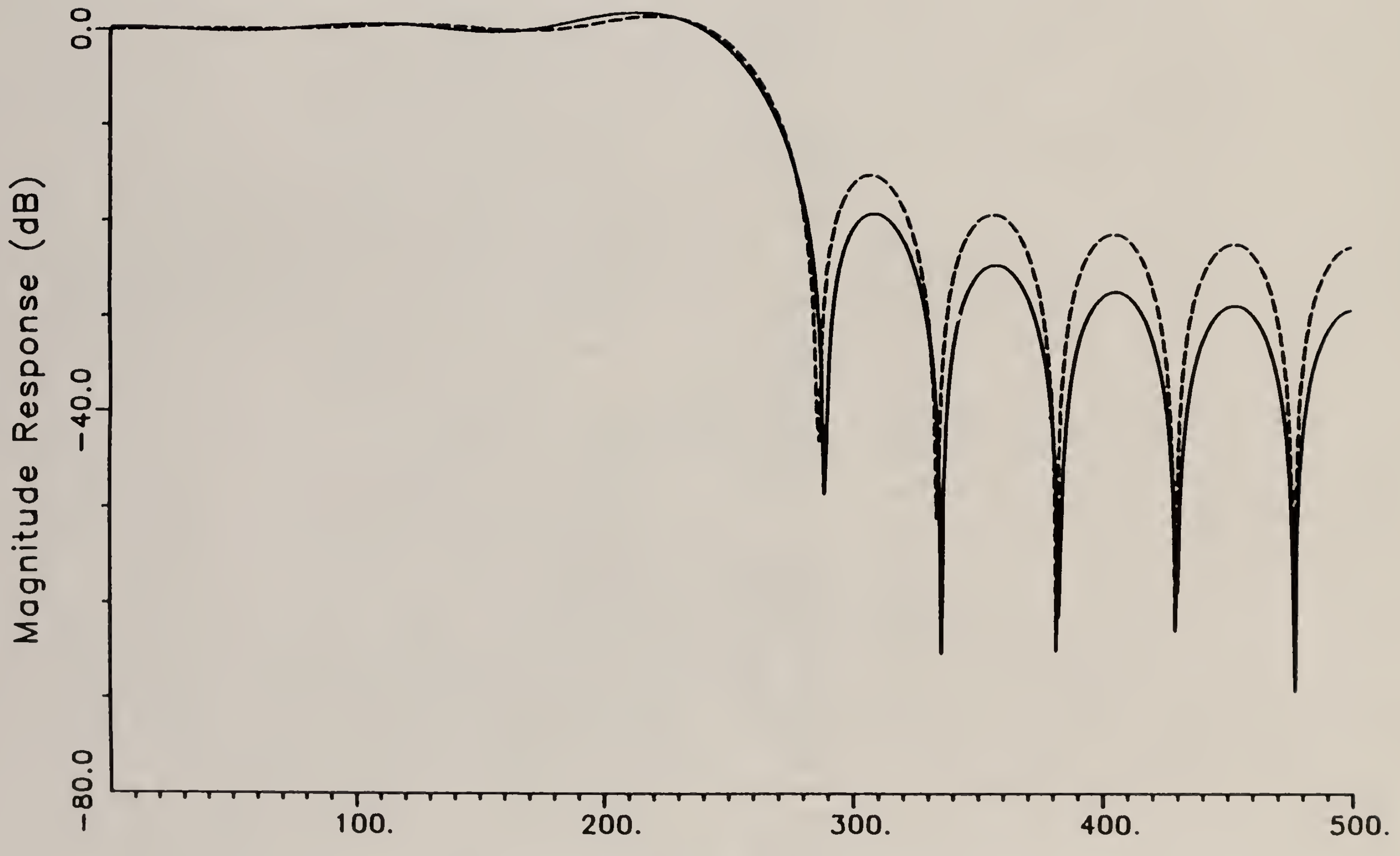

Figure 2. Example 1, comparing the new and frequency–sampling methods.

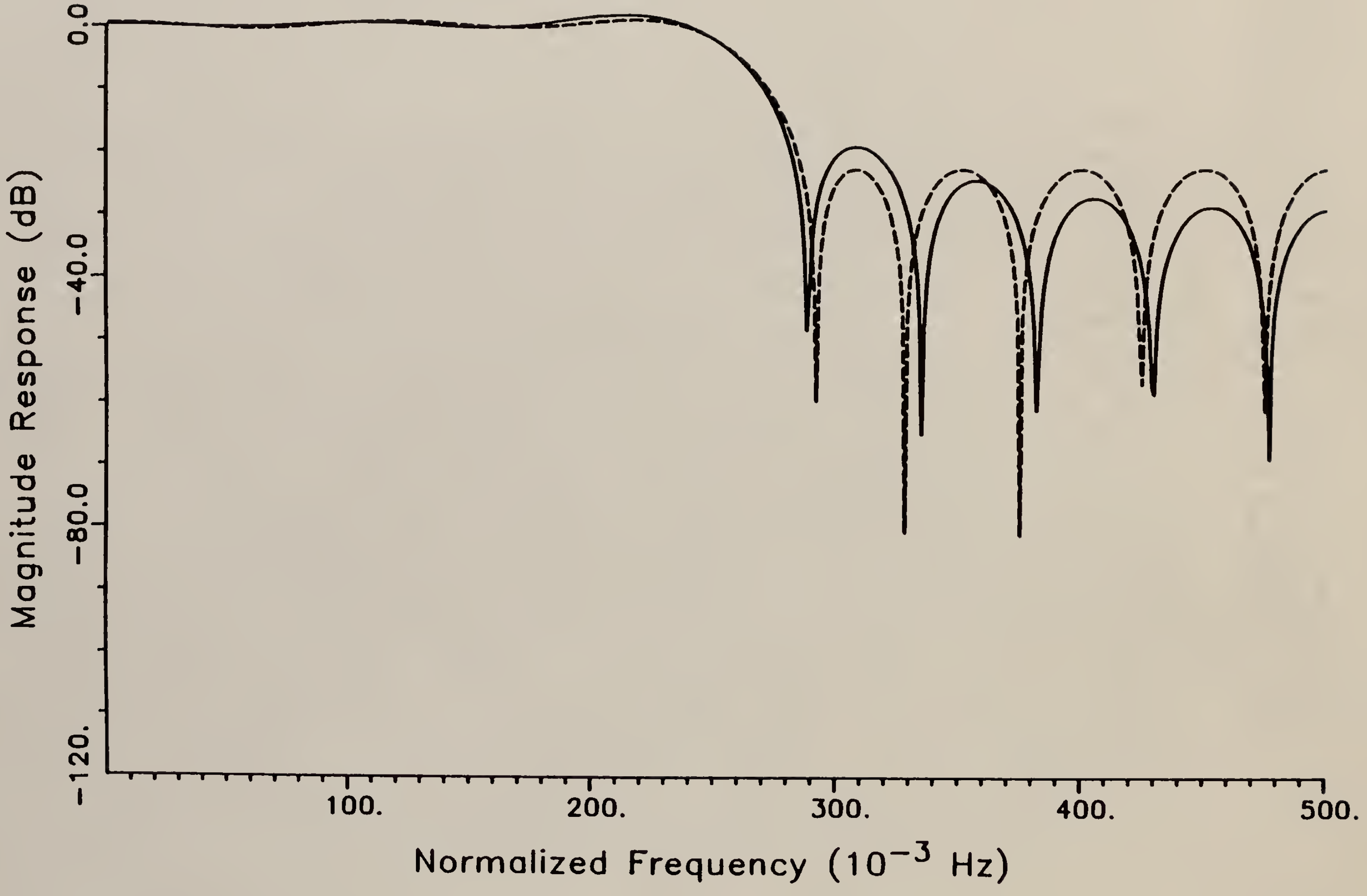

Figure 3. Example 1, comparing the new and Parks–McClellan methods.

Figures 4 and 5 represent LP filters of impulse length 95 defined over the subintervals $f \in [0,.2] \cup [.22,.5]$. Since the transition region is narrow, the mid-transition roll-off is very steep and the new methods advantages over the frequency-sampler are not decisive. Again, frequency-sampling provides a smoother pass-band roll-off but the new method provides about 5 dB more attenuation in the stop-band.

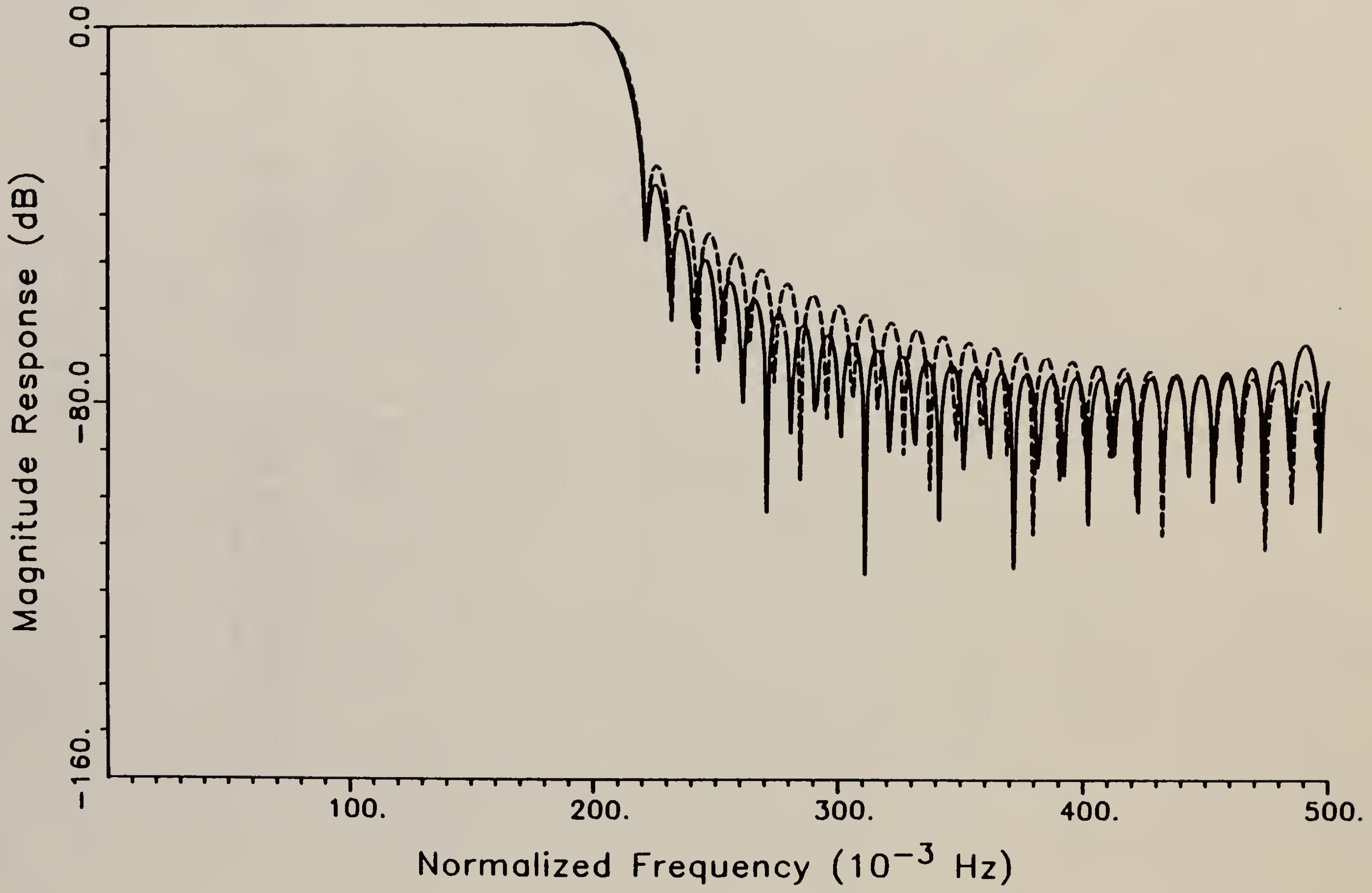

Figure 4. Example 2, comparing the new and frequency-sampling methods.

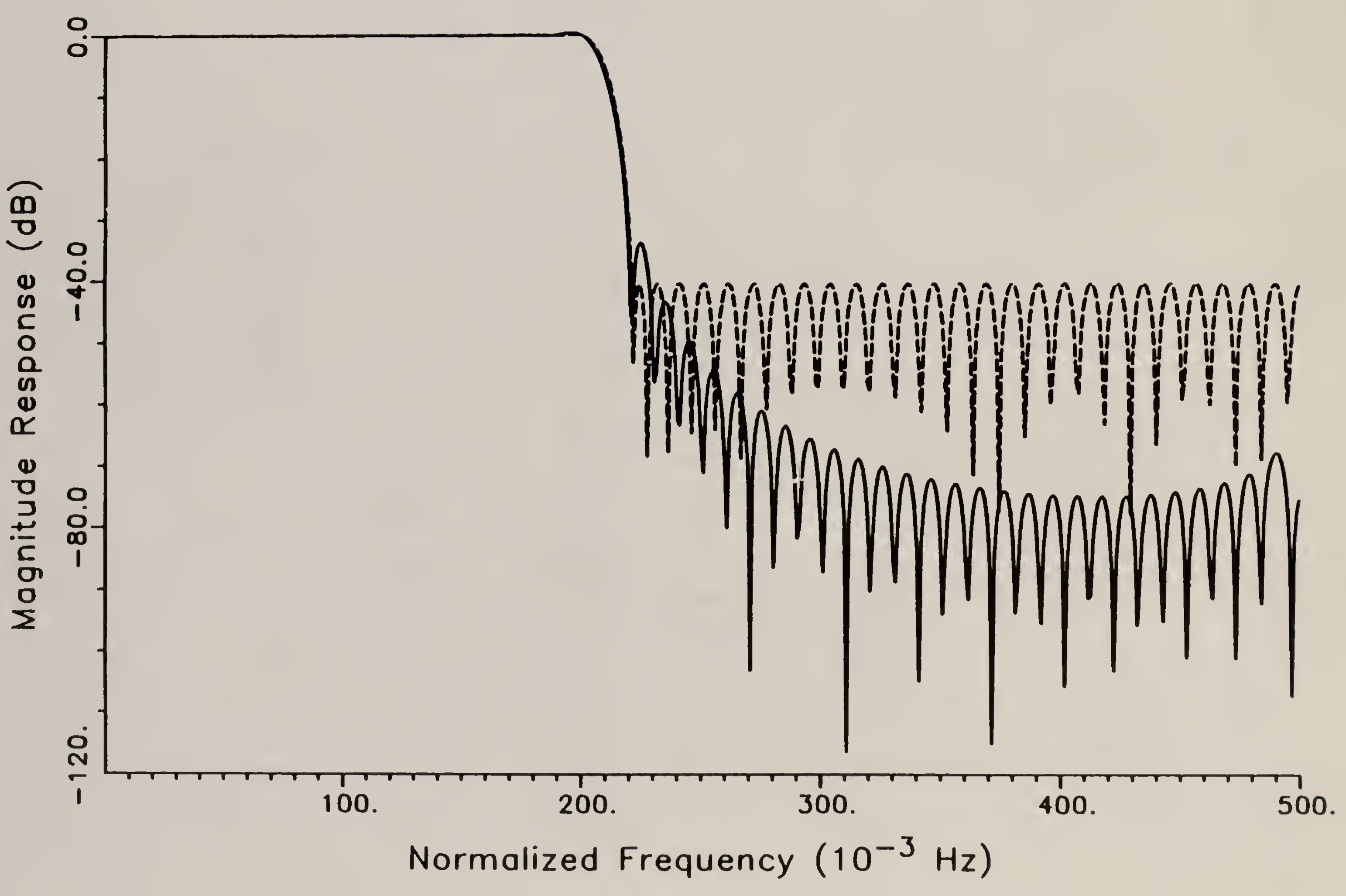

Figure 5. Example 2, comparing the new and Parks–McClellan methods.

Figures 6 and 7 represent LP filters of impulse length 95 defined on $f \in [0,.2] \cup [.25,.5]$. The transition width has been widened and the new method provides approximately 36 dB additional attenuation over frequency-sampling in the stop-band while still maintaining smoother pass-band rolloff characteristics. The Parks-McClellan method, as in all cases, provides superior performance in every respect.

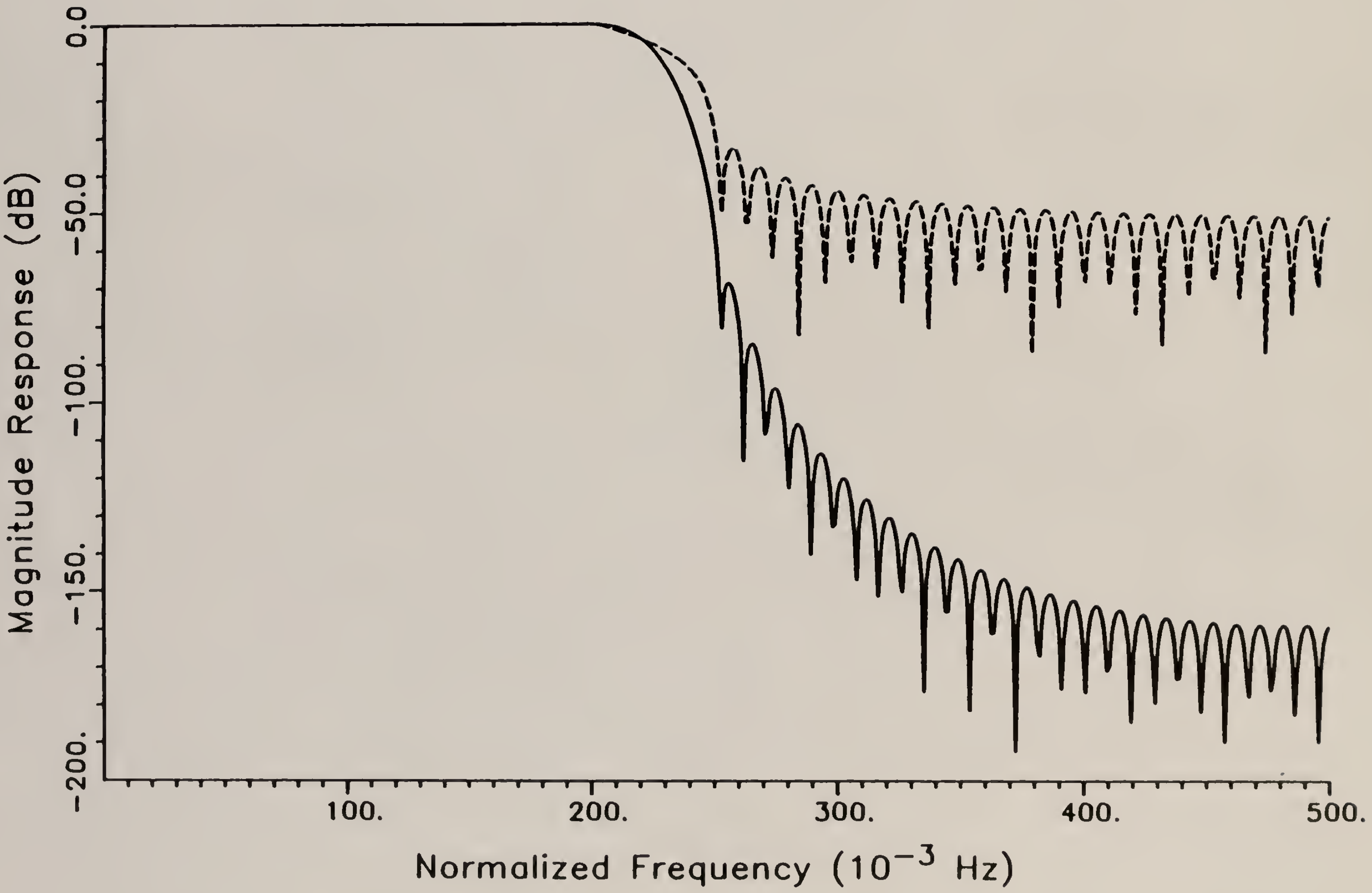

Figure 6. Example 3, comparing new and frequency-sampling methods.

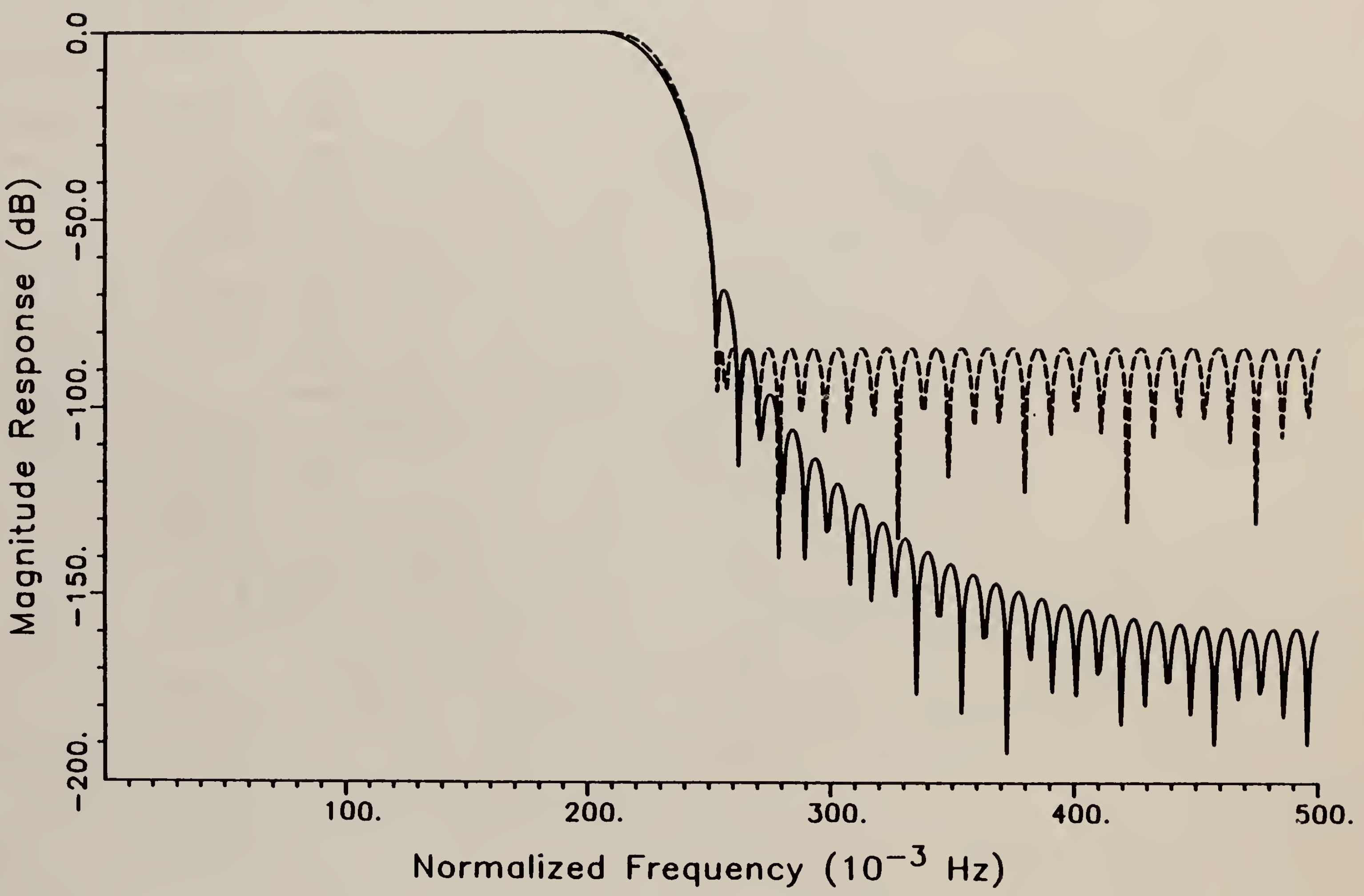

Figure 7. Example 3, comparing new and Parks–McClellan methods.

Figure 8 represents band-pass filters of impulse length 95 defined on the subintervals $f \in [0,.16] \cup [.2,.24] \cup [.28,.5]$ and compares an optimal solution (Parks-McClellan) to the new method. The new method sacrifices 6 dB to the optimal solution in the stop-band of the first transition region. However, it betters the optimal solution by 12 dB over the second transition.

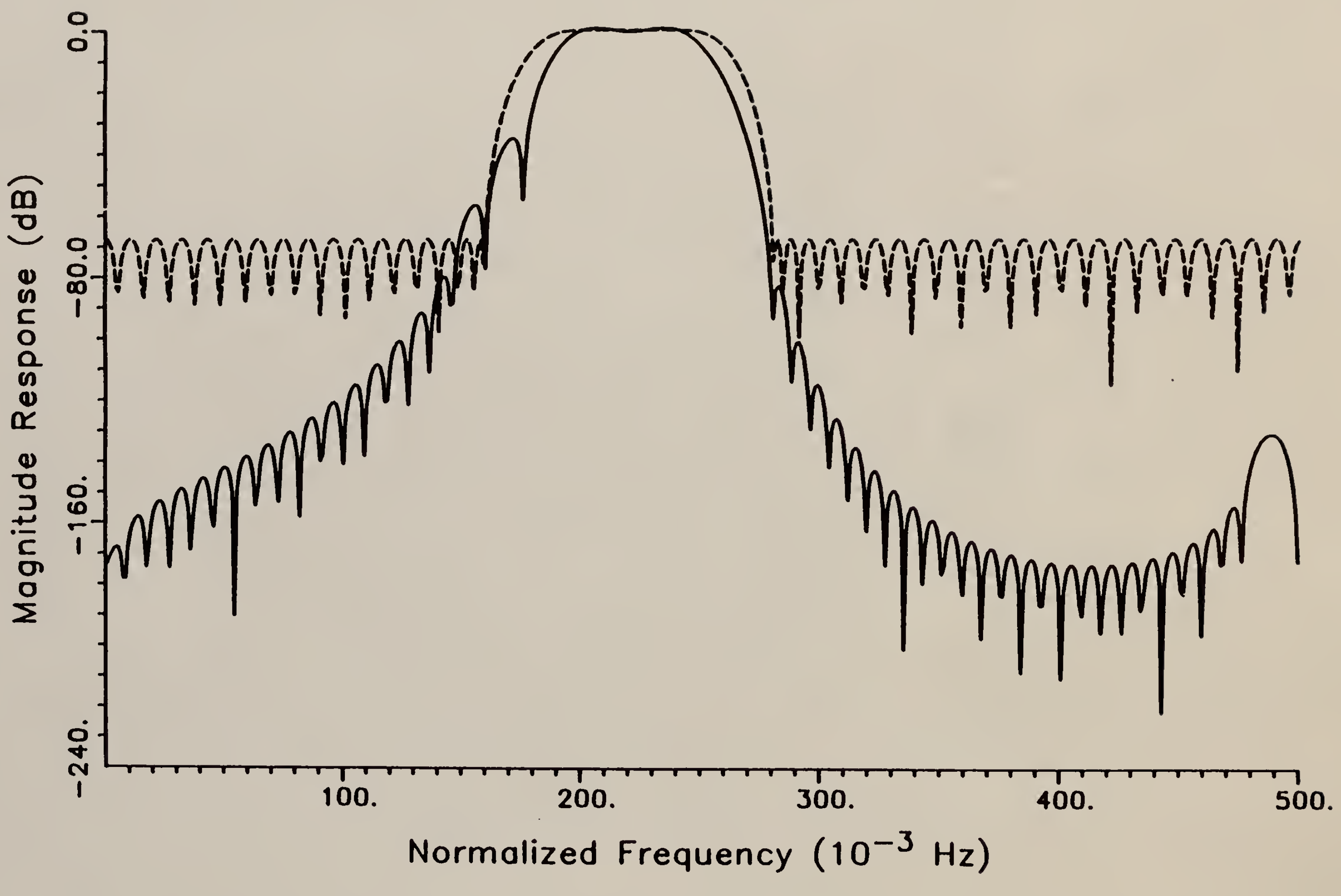

Figure 8. Example 4, comparing new and Parks–McClellan methods.

Comparing Some Design Times

All filters were designed on a 16 MHz Compaq DESKPRO running Borland's C compiler, version 1.5. Approximate times for filters with 95 impulse coefficients are as follows.

* Parks-McClellan Method (768 grid points):	120 sec.
* Frequency-sampling via DFT:	39 sec.
* New Method:	10 sec.
* Frequency-sampling via a "fast" DFT:	5 sec.

These times reflect the use of IEEE-standard floating-point emulation by the program. Addition of math-coprocessor support should substantially reduce the time needed to design a filter.

CHAPTER SEVEN

CONCLUSION FOR THE DESIGN OF FIR DIGITAL FILTERS

Introduction

A new method has been described for the design of FIR digital filters. This method has been found to be effective for designing large FIR digital filters of arbitrary response type while allowing control over transition band frequencies and band weighting.

Design Speed of New Method

It is extremely fast compared to the Remez exchange since Lagrange interpolation is performed over (N-1)/2 + 1 points as apposed to the Remez exchange which interpolates more than ten to twenty times that many points. This fact in conjunction with the iterative nature of optimal design processes can incur prohibitive design times for large filters, particularly if hardware floating-point support is not available. The gain in speed as compared to a frequency-sampling method is based on obtaining the impulse coefficients via an N/2-point DCT instead of an N-point IDFT.

Summary of Performance

Lagrange interpolation based on an equally spaced set of extremals is used to approximate the response of the filter. This forces some ripple to be induced into the pass and stop-bands thereby improving filter behavior at the band edges. These band-edge characteristics depend on the transition width and also on the general placement of the transition band with respect to the zero frequency. While it is not always possible to obtain a "usable" filter by using a one-pass algorithm, it becomes more likely when using the new design method. The advantages of the new method are most clearly realized when designing filters which do not impose extreme requirements on the transition regions. For example, to impose a transition width of [0.001] on the interval [0.0 , 0.5] of a 100-point filter would likely result in extreme behavior at the band edges. In this situation, a standard frequency-sampling realization would likely provide superior performance although it to would probably be unusable. The reason for the superiority in an extreme case such as this is due to the fact that

Lagrange interpolation enforces smooth transitions between bands. This results in an effective mid rolloff which is steeper than given by the frequency-sampling method even though the transition frequencies may by the same.

Problems With New Method

Problems are often encountered when designing with transition widths of greater than about [0.2] or less than [0.001] on the interval [0.0 , 0.5]. This is primarily due to the finite number of points in the dense frequency grid or roundoff error. This problem can be accentuated by very large filters which increase sensitivity to round-off in the Lagrange interpolation.

REFERENCES

1. McClellan, J. H., and Parks, T. W., "A Computer Program for Designing Optimum FIR Linear Phase Digital Filters", IEEE Trans. Audio Electroacoust., Vol. AU-21, No. 6, pp. 506-525, 1973.

2. Rabiner, L. R., and McClellan, J. H., and Parks, T. W., "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximation", Proc. IEEE, Vol. 63, No.4, pp. 595-609, 1975.

3. Oppenheim, A. V., and Schafer, R. W., "Digital Signal Processing", Prentice-Hall, Englewood Cliffs, N.J.,1975.

4. Ahmed, N., and Natarajan, T., "Discrete-Time Signals and Systems", Reston, Reston, Va., 1983.

5. Ahmed, N., and Rao, K. R., "Orthogonal Transforms for Digital Signal Processing", Springer-Verlag, New York, 1975.

6. Hamming, R. W., "Numerical Methods for Scientists and Engineers, McGraw Hill, New York, 1973.

APPENDIX

PROGRAM SOFTWARE LISTING

Introduction to Appendix

The C source files included in the appendix reflect the thread associated with the FIR filter design section only. This specifically includes the main program shell, all necessary header files (?.h), and all source files needed by the shell to edit a filter, compute the coefficients, and calculate the response. The peripheral routines such as plotting utilities and stack operations are not included. However, the primary register creation and maintenance routines needed by the program are included in case a programmer should wish to expand or revise the existing software.

Listing of Source Files

common.h	Standard header
screen.h	Screen macros
smath.h	Math definitions and macros
fir.h	Filter structures and definitions
rolf.h	Main program and register definitions
rolf.c	Main shell
filter.c	Choose FIR or IIR filter
firfilt.c	Edit, compute, I/O, display FIR filter
firparms.c	Edit FIR filter parameters
fircoef.c	Choose Parks-McClellan or new method
firsetup.c	Allocate memory, find desired response, estimate extremals, call Remez
remez.c	Remez exchange
makeres.c	Computes desired response from filter parameters
estextr.c	Estimate the initial set of extremals
findextr.c	Find all extremals
choosend.c	Select the correct extremals
estrho.c	Estimate the Chebyshev error
lagrange.c	Lagrange interpolation
alphas.c	Convert cosine to filter coefficients
firres.c	Compute the response from the coefficients
stackops.c	Program register creation and maintenance

```
/************************************************************
 *
 *  SOURCE FILE:      common.h
 *
 *
 *  FUNCTION:         None
 *
 *
 *  DESCRIPTION:      Provides some general declarations,
 *                    constants, and utilities.
 *
 *
 *  DOCUMENTATION
 *  FILES:            None
 *
 *
 *  ARGUMENTS:        None
 *
 *
 *  RETURN:           None
 *
 *
 *  FUNCTIONS
 *  CALLED:           None
 *
 *
 *  AUTHOR:           Scott A. Nichols
 *
 *
 *  DATE CREATED:     1Oct87
 *
 *
 ***********************************************************/

#ifndef _COMMON_H
#define _COMMON_H

typedef int BOOL;  /* boolean data type                    */


/*---------------------------------------------------------*/
/*  Define some termination and test constants.            */
/*---------------------------------------------------------*/
#define FALSE          0
#define TRUE           !FALSE
#define FAIL           0
#define SUCCEED        !FAIL
#define OFF            0
#define ON             !OFF
```

```
#define EXIT            0
#define OK              0
#define ERROR           !OK


/*-------------------------------------------------------------*/
/* Define some useful macros.                                  */
/*-------------------------------------------------------------*/
#define FOREVER         for (;;)
#define DIM(x)          (sizeof(x)/sizeof(x[0]))
#define SWAP(x,y,t)     ((t = (y), (y) = (x), (x) = (t))
#define YES(ch)         (ch == 'y' || ch == 'Y')
#define NO(ch)          (ch == 'n' || ch == 'N')
#define STACKROLL(stk,pntr,end) \
                  (pntr = (stk[pntr+1] == end ? 0 : ++pntr))
#define INRANGE(low, x, up) ((x) >= low && (x) <= up)

#endif
```

```
/**************************************************************
 *
 *
 *   SOURCE FILE:      screen.h
 *
 *
 *   FUNCTION:         None
 *
 *
 *   USAGE:            NA
 *
 *
 *   DESCRIPTION:      Definition and declaration for some
 *                     constants and functions used in screen
 *                     I/O.
 *
 *
 *   DOCUMENTATION
 *   FILES:            None
 *
 *
 *   ARGUMENTS:        NA
 *
 *
 *   RETURN:           NA
 *
 *
 *   FUNCTIONS
 *   CALLED:           NA
 *
 *
 *   AUTHOR:           Scott A. Nichols
 *
 *
 *   DATE CREATED:     4Dec88
 *
 *
 *   REVISIONS:        Ver 1.00
 *
 *
 **************************************************************/

#ifndef _SCREEN_H
#define _SCREEN_H

#define  HERC_BASE  0xB000
#define  CGA_BASE   0xB800
#define  SCRN_BASE  HERC_BASE
```

```
/*-------------------------------------------------------------*/
/*  Define some ansi cursor and screen functions.              */
/*-------------------------------------------------------------*/
#define save_cursor()       (fputs("\033[s", stdout))
#define restore_cursor()    (fputs("\033[u", stdout))
#define clear_screen()      (fputs("\033[2J", stdout))
#define right_cursor(x)     (fputs("\033[" x "C", stdout))
#define left_cursor(x)      (fputs("\033[" x "D",stdout))
#define up_cursor(y)        (fputs("\033[" y "A",stdout))
#define down_cursor(y)      (fputs("\033[" y "B", stdout))
#define clear_line()        (fputs("\033[K", stdout))
#define move_cursor(y_x)    (fputs("\33[" y_x "H", stdout))


/*-------------------------------------------------------------*/
/*  Define some screen interface functions.                    */
/*-------------------------------------------------------------*/
char    query(char *string);
void    key_pressed(void);
double  getd(void);

#endif
```

```
/***********************************************************
 *
 *
 *  SOURCE FILE:      smath.h
 *
 *
 *  FUNCTION:         None
 *
 *
 *  DESCRIPTION:      Some commonly used math functions
 *                    defined as macros, structure
 *                    definitions for complex, polynomial,
 *                    and rational data objects, and some
 *                    common mathematical constants.
 *
 *
 *  DOCUMENTATION
 *  FILES:            None
 *
 *
 *  ARGUMENTS:        Not Applicable
 *
 *
 *
 *  RETURN:           Not Applicable
 *
 *
 *  FUNCTIONS
 *  CALLED:           None
 *
 *
 *  AUTHOR:           Scott A. Nichols
 *
 *
 *  DATE CREATED:     24Jan87
 *
 *
 *  REVISIONS:        Ver 1.00
 *
 *
 ***********************************************************/

#ifndef _SMATH_H
#define _SMATH_H

#ifndef  _MATH_H
#include <math.h>
#endif

#define MAXLEN        4000  /* maximum length of a complex
```

```
                                 array under MS DOS.         */


/*-------------------------------------------------------*/
/*  Define some useful mathematical macros.              */
/*-------------------------------------------------------*/
#define SIGN(x)      ((x) < 0 ? -1:((x) > 0 ? 1:0))
#define LOG2(x)      (log(x)/M_LOG2)
#define ABS(x)       ((x) < 0 ? -(x):(x))
#define FRAC(x)      (fabs(x)-(int)(x))
#define ROUND(x)   \
          (FRAC(x) < 0.5 ? (int)(x) : (int)(x) + SIGN(x))
#define ODD(x)       ((x)%2 == 0 ? 0:1)
#define MIN(x,y)     ((x) <= (y) ? (x):(y))
#define MAX(x,y)     ((x) >= (y) ? (x):(y))
#define SQR(x)       ((x)*(x))
#define ISPOW2(x)    (!FRAC(LOG2(x)))
#define NEXTPOW2(x) \
      (ISPOW2(x) ? x : ROUND(pow(2, (int)(LOG2(x)) + 1)))


/*-------------------------------------------------------*/
/* Template for a data object of type complex.           */
/*-------------------------------------------------------*/
typedef struct
    {
    double re;
    double im;
    } COMPLEX;



#define P_MAX_DEG   20   /* maximum degree of polynomial */
#define P_MAX_COEF  P_MAX_DEG + 1 /* max number of coef  */


/*-------------------------------------------------------*/
/* Template for an object containing the coefficients    */
/* of a polynomial.                                      */
/*-------------------------------------------------------*/
typedef struct
    {
    int deg;

    double  p[P_MAX_COEF];
    } POLYNOMIAL;


/*-------------------------------------------------------*/
/* Template for an object containing the coefficients    */
/* of a rational function.                                     */
/*-------------------------------------------------------------*/
typedef struct
    {
    POLYNOMIAL num;
    POLYNOMIAL den;
    } RATIONAL;


/*-------------------------------------------------------------*/
/* Constants defining data types and formats.                  */
/*-------------------------------------------------------------*/
#define REAL    1
#define CMPLX   2
#define DEG     3
#define RAD     4
#define HZ      5
#define RECT    6
#define POLAR   7
#define MAG     8
#define LOGMAG  9
#define PHASE   10


/*-------------------------------------------------------------*/
/* Some common mathematical constants.                         */
/*-------------------------------------------------------------*/
#define M_E             2.71828182845904524
#define M_LOG2E         1.44269504088896341
#define M_LOG2          (1/M_LOG2E)
#define M_LOG10E        0.434294481903251828
#define M_LN2           0.693147180559945309
#define M_LN10          2.30258509299404568
#define M_PI            3.14159265358979324
#define M_2PI           6.28318530717958448
#define M_PI_2          1.57079632679489662
#define M_PI_4          0.785398163397448310
#define M_1_PI          0.318309886183790672
#define M_2_PI          0.636619772367581343
#define M_1_SQRTPI      0.564189583547756287
#define M_2_SQRTPI      1.12837916709551257
#define M_SQRT2         1.41421356237309505
#define M_SQRT_2        0.707106781186547524

#endif
```

```
/**************************************************************
 *
 *
 *  SOURCE FILE:   fir.h
 *
 *
 *  FUNCTION:      NA
 *
 *
 *  DESCRIPTION:   Definitions, declarations, and error
 *                 constants for the design of fir filters.
 *
 *
 *  DOCUMENTATION
 *  FILES:         None
 *
 *
 *  ARGUMENTS:     NA
 *
 *
 *  RETURN:        NA
 *
 *
 *  FUNCTIONS
 *  CALLED:        NA
 *
 *
 *  AUTHOR:        Scott A. Nichols
 *
 *
 *  DATE CREATED:  13Nov87
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *************************************************************/

#ifndef  _FIR_H
#define  _FIR_H

#define PARKS_McCLELLAN 1
#define NEW_METHOD      2

#define LOWPASS         1
#define HIGHPASS        2
#define BANDPASS        3
#define BANDSTOP        4

#define  MAX_ITER       25 /* maximum number of iterations
                              in the Remez exchange       */
```

```
#define  NOBANDS         5  /* number of distinct bands in
                               the filter                      */
#define  BRICKWALL       1  /* brickwall magnitude res-
                               ponse type                      */
#define  GENERAL         2  /* arbitrary response type         */
#define  EVEN_SYMMETRY   1  /* even-symmetry coefficients      */
#define  ODD_SYMMETRY    2  /* odd-symmetry coefficients       */


/*-----------------------------------------------------------------*/
/* Template for object containing the index into the          */
/* array of transition frequencies.                            */
/*-----------------------------------------------------------------*/
typedef struct
    {
    int lw; /* lower transition frequency index                */
    int up; /* upper transition frequency index                */
    } TRAN_INDEXES;


/*-----------------------------------------------------------------*/
/*  Template for object containing band transition            */
/*  frequencies.                                               */
/*-----------------------------------------------------------------*/
typedef struct
    {
    double lw;
    double up;
    } FREQUENCIES;


/*-----------------------------------------------------------------*/
/*  Template for the object containing the parameters of      */
/*  a particular FIR type digital filter.                      */
/*-----------------------------------------------------------------*/
typedef struct
    {
    int  type;      /* even or odd symmetry filter coef       */
    int  response;  /* brickwall or arbitrary mag res         */
    int  order;     /* number of impulse coefficients         */
    int  nobands;   /* number of distinct filter bands        */

    double band_value[NOBANDS];  /* magnitude value in
                                    a filter band              */
    double band_weight[NOBANDS]; /* error weighting in a
                                    filter band                */

    FREQUENCIES tran_freq[NOBANDS]; /* array of all tran-
                                       sition frequencies */
    } FIR_SPECS;
```

```
/*---------------------------------------------------------*/
/* Some function definitions used by the FIR  design      */
/* utilities.                                              */
/*---------------------------------------------------------*/
extern int     fir_parms();
extern int     remez();
extern int     make_response();
extern int     estimate_extremals();
extern int     find_extremals();
extern int     choose_endpoints();
extern int     alphas();
extern double  lagrange();
extern double  estimate_rho();

#endif
```

```
/************************************************************
 *
 *
 *  SOURCE FILE:      rolf.h
 *
 *
 *  FUNCTION:         None
 *
 *
 *  DESCRIPTION:
 *
 *      Definitions, declarations, and constants
 *      for the rolf specific functions and
 *      operations.
 *
 *
 *  DOCUMENTATION
 *  FILES:             A NEW METHOD FOR THE DESIGN
 *                     OF FIR DIGITAL FILTERS
 *
 *
 *  ARGUMENTS:        NA
 *
 *
 *  RETURN:           NA
 *
 *
 *  FUNCTIONS
 *  CALLED:           NA
 *
 *
 *  AUTHOR:           Scott A. Nichols
 *
 *
 *  DATE CREATED:     16May88
 *
 *
 *  REVISIONS:        Ver 1.00
 *
 ***********************************************************/

#ifndef  _ROLF_H
#define  _ROLF_H

#include "common.h"
#include "screen.h"
#include "smath.h"

#define  REGLEN        1024  /* default register length   */
#define  PLOTLEN       1024  /* maximum number of plotting
```

```
                                    points                    */

#define  READ   0  /* signifies a read from the disk       */
#define  WRITE  1  /* signifies a write to the disk        */


/*---------------------------------------------------------*/
/* These are the registers currently created within rolf. */
/*---------------------------------------------------------*/
#define  X              1
#define  Y              2
#define  Z              3
#define  T              4
#define  R0             5
#define  WORK           6


/*---------------------------------------------------------*/
/* These are register contents or last-action indicators. */
/*---------------------------------------------------------*/
#define NO_CONTENTS         0
#define FIR_COEFFICIENTS    1
#define IIR_COEFFICIENTS    2
#define FIR_RESPONSE        3
#define IIR_RESPONSE        4
#define GENERAL_DATA        5
#define TRANS_DATA          6
#define INV_TRANS_DATA      7



/*---------------------------------------------------------*/
/* Structure template for the internal rolf registers.    */
/*---------------------------------------------------------*/
typedef struct
    {
   int len;            /* register length                  */
   int mode;           /* rectangular or polar             */
   int     type;       /* real or complex data type        */
   int     contents;   /* index of last-action or
                                         contents message */

    COMPLEX *reg;  /* pointer to a complex data array      */
   } REG;



/*---------------------------------------------------------*/
/* Declaration of functions invoked within rolf.          */
/*---------------------------------------------------------*/
```

```
extern int      create_regs();
extern REG      *get_reg();
extern void     free_regs();
extern void     roll_up();
extern void     roll_down();
extern void     enter_reg();
extern void     switch_reg();
extern char     *toggle_lock();
extern void     disable_roll();
extern void     enable_roll();
extern void     clear_reg(REG *);
extern void     copy_reg(REG *, REG *);
extern void     clear_stack();
extern void     bin_stk_fix();

#endif
```

```
/*************************************************************
 *
 *
 *  SOURCE FILE:   rolf.c
 *
 *
 *  FUNCTION:      main()
 *
 *
 *  ARGUMENTS:     None
 *
 *
 *  RETURN:        int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      This is the outermost shell of rolf.exe and
 *      is used to set up the key parameters of the
 *      program at runtime.  These include adjusting
 *      the stack length, creating and clearing all
 *      the internal stack and storage registers,
 *      and displaying the menu of primary program
 *      functions.  These functions include the
 *      following: Various utilities which generate
 *      data; unary and binary (including trig)
 *      operations on the stack registers; register
 *      operation such as rectangular-to-polar and
 *      storing a register to disk; both IIR and FIR
 *      filter design; signal processing such as FFT's
 *      and IFFT's;  numeric and graphical screen
 *      output of register contents; exit to DOS
 *      shell; stack register manipulations such as
 *      swap X and Y, rotate up and down, and others.
 *
 *
 *  DOCUMENTATION
 *  FILES:         A NEW METHOD FOR THE DESIGN
 *                 OF FIR DIGITAL FILTERS
 *
 *
 *  CONSTANTS:     OK      common.h
 *                 EXIT    common.h
 *                 REGLEN  rolf.h
 *                 R0      rolf.h
 *
 *
 *  MACROS
 *  EXPANDED:      YES()     common.h
 *                 FOREVER   common.h
 *
 *
 *  FUNCTIONS
 *  CALLED:          create_regs()   stackops.c
 *                   clear_stack()   stackops.c
 *                   free_regs()     stackops.c
 *                   clear_reg()     stackops.c
 *                   disp_stack()    stackops.c
 *                   clear_screen()  screen.h
 *                   data_gen()      datagen.c
 *                   arithops()      arithops.c
 *                   reg_ops()       regops.c
 *                   filter()        filter.c
 *                   sigproc()       sigproc.c
 *                   scrn()          scrn.c
 *                   system()        DOS kernel
 *                   stack()         stack.c
 *                   query()         screen.c
 *
 *
 *
 *  AUTHOR:          Scott A. Nichols
 *
 *
 *  DATE CREATED:    18May88
 *
 *
 *  REVISIONS:       Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include "rolf.h"

int reg_len = REGLEN;

unsigned stklen; /* defined by the Borland startup code   */

main()
{
    char ch;

    stklen = 8000;

/*- create the stack and storage registers ---------------*/
    if (create_regs())
        {
```

```
        fputs("Can't create the stack registers",stderr);
        exit(2);
        }

/*- clear the stack and storage registers ----------------*/
    clear_stack();
    clear_reg(get_reg(R0));

    FOREVER
        {
        clear_screen();
        disp_stack(); /* displays the status of the stack */
        printf("\n0:  Exit Program");
        printf("\n1:  Data generation");
        printf("\n2:  Arithmetic operations");
        printf("\n3:  Register operations");
        printf("\n4:  Filter design");
        printf("\n5:  Signal processing");
        printf("\n6:  Plotting/Printing");
        printf("\n7:  DOS shell");
        printf("\n    <RETURN> Stack");
        printf("\n\n>>  ");

        switch (query("") - '0')
            {
            case EXIT:
/*------------- double check and free rolf's registers ---*/
                clear_screen();
                ch = query("Exiting Program. Continue ? ");
                if (YES(ch))
                    {
                    free_regs();
                    return(OK);
                    }
                continue;

            case 1:
/*------------- invoke the data generation utilities -----*/
                if (data_gen())
                    query("Error in data generation");
                break;

            case 2:
/*------------- invoke arithmetic operation utilities ----*/
                if (arithops())
                    query("Error in arithmetic ops");
                break;

            case 3:
/*------------ invoke the stack register operations ------*/
```

```
                    if (reg_ops())
                         query("Error in register operations");
                     break;

                case 4:
/*-------------- invoke the filter design utilities -------*/
                     if (filter())
                         query("Error in filter routine");
                     break;

                case 5:
/*-------------- invoke the signal processing utilities ---*/
                     if (sigproc())
                         query("Error in signal processing");
                     break;

                case 6:
/*-------------- invoke the screen output function  -------*/
                     if (scrn())
                         query("Error in screen io");
                     break;

                case 7:
/*-------------- load a copy of command.com and execute ---*/
                     if (system("command.com") != 0)
                     fputs("\nEXEC of COMMAND.COM failed\n",
                         stderr);
                     break;

                default:
/*-------------- invoke the stack manipulation utility ----*/
                     if (stack())
                         query("Error in stack manipulations");
                     break;
                } /* end switch (menu) */

            } /* end FOREVER */

    } /* end main */
```

```
/************************************************************
 *
 *
 *   SOURCE FILE:   filter.c
 *
 *
 *   FUNCTION:      int filter(void)
 *
 *
 *   DESCRIPTION:
 *
 *       This function displays the menu allowing
 *       selection between the design of an IIR or
 *       FIR digital filter.  After a selection is
 *       is made, the appropriate function is invoked.

 *
 *
 *   DOCUMENTATION
 *   FILES:         None
 *
 *
 *   ARGUMENTS:     None
 *
 *
 *   RETURN:        int: OK
 *
 *
 *   CONSTANTS:     OK  common.h
 *
 *
 *   MACROS
 *   EXPANDED:      FOREVER   common.h
 *
 *
 *   FUNCTIONS
 *   CALLED:        iir_filter()    iirfilt.c
 *                  fir_filter()    firfilt.c
 *                  query()         screen.c
 *                  clear_screen()  screen.h
 *
 *
 *   AUTHOR:        Scott A. Nichols
 *
 *
 *   DATE CREATED:  17May88
 *
 *
 *   REVISIONS:     Ver 1.00
 *
 *
 *****************************************************************/

#include <stdio.h>
#include "rolf.h"

int filter()
{
    char   ch;

    FOREVER
        {
        clear_screen();
        printf("\n0:  Exit");
        printf("\n1:  FIR design");
        printf("\n2:  IIR design");
        printf("\n\n>>  ");

        switch (query("") - '0')
            {
            case EXIT:
                return(OK);

            case 1:
/*------------- design an FIR digital filter -------------*/
                if (fir_filter())
                    query("Error in FIR design");
                break;

            case 2:
/*------------- design an IIR digital filter -------------*/
                if (iir_filter())
                    query("Error in IIR design");
                break;

            default:
                break;
            } /* end switch (menu) */

        } /* end FOREVER */

} /* end filter() */
```

```
/*****************************************************************
 *
 *
 *  SOURCE FILE:    firfilt.c
 *
 *
 *  FUNCTION:       int fir_filter(void)
 *
 *
 *  ARGUMENTS:      None
 *
 *
 *  RETURN:         int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      This function defines the objects containing
 *      the FIR filter parameters and transition
 *      frequency indexes. It also provides the menu
 *      and permits selection of the following filter
 *      utilities: Editing filter parameters;
 *      computing filter coefficients; computing
 *      a simulation of the filter response;
 *      displaying the impulse coefficients to
 *      the CRT; reading an FIR filter from the
 *      disk;  writing an FIR filter to the disk;
 *      displaying the FIR filter parameters;
 *      manipulating the stack registers.
 *
 *
 *  DOCUMENTATION
 *  FILES:          A NEW METHOD FOR THE DESIGN OF
 *                  FIR DIGITAL FILTERS
 *
 *
 *  CONSTANTS:      OK                  common.h
 *                  EXIT                common.h
 *                  READ                rolf.h
 *                  WRITE               rolf.h
 *                  FIR_COEFFICIENTS    rolf.h
 *                  X                   rolf.h
 *
 *
 *  MACROS
 *  EXPANDED:       FOREVER   common.h
 *
 *
 *  FUNCTIONS
 *  CALLED:         get_reg()       stackops.c
 *                  enter_reg()      stackops.c
 *                  disp_stack()     stackops.c
 *                  fir_parms()      firparms.c
 *                  fir_coef()       fircoef.c
 *                  clear_screen()   screen.h
 *                  query()          screen.c
 *                  fir_response()   firres.c
 *                  fir_display()    firdisp.c
 *                  fir_io()         firio.c
 *                  stack()          stack.c
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include "rolf.h"
#include "fir.h"

/* template in fir.h, this object contains the filter
   parameters, initialization is to indicate that a
   filter is not present                                 */
FIR_SPECS fir = {-1, -1};

/* template in fir.h, this object contains the indexes
   of the transition frequencies                         */
TRAN_INDEXES tran_index[NOBANDS];

double rho; /* estimate of the Chebyshev error           */


int fir_filter()
{
    REG *x;  /* template in rolf.h, defines a pointer to
                an internal stack or storage register    */
    int i;

    x = get_reg(X); /* set the pointer to the X register  */

    FOREVER
        {
        clear_screen();
        disp_stack();
        printf("\n0:  Exit");
        printf("\n1:  Edit  filter parameters");
        printf("\n2:  Compute impulse coefficients");
        printf("\n3:  Compute filter response");
        printf("\n4:  Display filter coefficients");
        printf("\n5:  Read filter from disk");
        printf("\n6:  Write filter to disk");
        printf("\n7:  Display filter parameters");
        printf("\n    <RETURN> Stack");
        printf("\n\n>>  ");
        switch (query("") - '0')
            {
            case EXIT:
                return(OK);

            case 1:
/*------------- edit the FIR filter parameters -----------*/
                if (fir_parms())
                    query("Error entering filter");
                break;

            case 2:
/*------------- compute the fir impulse coefficients -----*/
                if (fircoef())
                    query("Error calc coef");
                break;

            case 3:
/*--------------------------------------------------------*/
/*              simulate the magnitude response           */
/*              of the filter                             */
/*--------------------------------------------------------*/
                clear_screen();
                if (x->contents != FIR_COEFFICIENTS)
                    {
                    query("Error: need filter
                           coefficients ");
                    break;
                    }
                enter_reg(); /* perform an RPN enter fnc  */
                if (fir_response())
                    query("Error calculating
                           filter response");
                break;

            case 4:
/*--------------------------------------------------------*/
/*              display the FIR impulse coefficients      */
/*              to the CRT                                */
/*-------------------------------------------------------------*/
                clear_screen();
                for (i = 0; i <= x->len-1; ++i)
                    {
                    if (i % 23 == 0 && i > 0)
                        if (query("\nQuit ?") == 'y')
                            break;
                    printf("\ncoef[%d] = %lf",i,
                        x->reg[i].re);
                    }
                query("\nPress any key to continue");
                continue;

            case 5:
/*-------------- read filter parameters from the disk -----*/
                if (fir_io(READ))
                    query("Error reading disk");
                continue;

            case 6:
/*-------------- write filter parameters to the disk ------*/
                if (fir_io(WRITE))
                    query("Error writing disk");
                continue;

            case 7:
/*-------------- Display the filter parameters ------------*/
                if (fir_display())
                    query("Error: displaying
                           filter parameters ");
                continue;

            default:
/*-------------- manipulate the stack registers -----------*/
                if (stack())
                    query("Error in stack ops");
                continue;
            } /* end switch (menu) */

        } /* end FOREVER */

} /* end fir_filter() */
```

```
/*****************************************************************
 *
 *
 *   SOURCE FILE:    firparms.c
 *
 *
 *   FUNCTION:       int fir_parms()
 *
 *
 *   ARGUMENTS:      None
 *
 *
 *   RETURN:         int: OK
 *
 *
 *   DESCRIPTION:
 *
 *       This function  prompts the user for the
 *       various FIR filter parameters which specify
 *       the response characteristics.  These are as
 *       follows:  Impulse response length; even or
 *       odd symmetry impulse coefficients; brickwall
 *       or general (arbitrary) magnitude response.
 *       If the magnitude response type is brickwall,
 *       then the number of distinct bands, amplification,
 *       Chebyshev error weighting, and transition
 *       frequencies for each band are also prompted for.
 *
 *
 *   DOCUMENTATION
 *   FILES:          A NEW METHOD FOR THE DISIGN
 *                   OF FIR DIGITAL FILTERS
 *
 *
 *   CONSTANTS:      OK          common.h
 *                   BRICKWALL   fir.h
 *
 *   MACROS
 *   EXPANDED:       None
 *
 *
 *   FUNCTIONS
 *   CALLED:         clear_screen()   screen.c
 *                   getd()           screen.c
 *
 *
 *   AUTHOR:         Scott A. Nichols
 *
 *
 *   DATE CREATED:   17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 **********************************************************/

#include <stdio.h>
#include "common.h"
#include "screen.h"
#include "fir.h"

/* template defined in fir.h, this object contains all
   pertinent parameters which determine the character-
   istics of the FIR filter                                  */
extern FIR_SPECS fir;

int  fir_parms()
{
    int i;

    clear_screen();
    printf("1: BrickWall\n");
    printf("2: General\n");
    printf("\nEnter choice > ");
    scanf("%d",&fir.response);

    clear_screen();
    printf("\n1: Symmetrical");
    printf("\n2: Asymmetrical\n");
    printf("\nEnter choice > ");
    scanf("%d",&fir.type);

    clear_screen();
    printf("\nFilter Length: > ");
    scanf("%d",&fir.order);

    clear_screen();
    if (fir.response == BRICKWALL)
        {
        printf("\nEnter The Number Of Distinct Bands > ");
        scanf("%d",&fir.nobands);
        printf("\n");

        clear_screen();
        for (i = 0; i < fir.nobands; ++i)
            {
            printf("\n\nEnter magnitude   for band[%d] > ",
                i+1);
            fir.band_value[i] = getd();
            printf("\nEnter band_weight for band[%d] > ",
                i+1);
            fir.band_weight[i] = getd();
            }

        clear_screen();
        for (i = 0; i < fir.nobands; ++i)
            {
            if (i == 0) /* this frequency is always zero  */
                fir.tran_freq[i].lw = 0;
            else
                {
                printf("\n\nEnter lower transition for"
                    " band[%d] > ",i+1);
                fir.tran_freq[i].lw = getd();
                }

/*--------- 0.5 corresponds to the Nyquist frequency -----*/
            if (i == fir.nobands-1)
                fir.tran_freq[i].up = 0.5;
            else
                {
                printf("\nEnter upper transition for"
                   " band[%d] > ",i+1);
                fir.tran_freq[i].up = getd();
                }
            }
        }

    clear_screen();
    return(OK);

} /* end fir_parms() */
```

```
/*****************************************************************
 *
 *
 *  SOURCE FILE:   fircoef.c
 *
 *
 *  FUNCTION:      int fir_coef()
 *
 *
 *  ARGUMENTS:     None
 *
 *
 *  RETURN:        int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      The primary purpose of this function is to
 *      invoke the appropriate routines which will
 *      compute the FIR filter coefficients.  The
 *      two choices offered are by the new method
 *      and by the Parks-McClellan method.  It also
 *      provides a utility which allows the density
 *      of the frequency grid to be adjusted.  This
 *      is especially important when using the Parks-
 *      McClellan method.  A higher density (larger
 *      value) means that the approximation will be
 *      more accurate but will take significantly more
 *      time to compute.  The stack manipulation
 *      function can also be invoked from this menu.
 *
 *
 *  DOCUMENTATION
 *  FILES:         A NEW METHOD FOR THE DESIGN
 *                 OF FIR DIGITAL FILTERS
 *
 *
 *  CONSTANTS:     OK                  common.h
 *                 EXIT                common.h
 *                 REAL                smath.h
 *                 PARKS_McCLELLAN     fir.h
 *                 NEW_METHOD          fir.h
 *                 FIR_COEFFICIENTS    rolf.h
 *
 *  MACROS
 *  EXPANDED:      FOREVER          common.h
 *                 STACKROLL()      common.h
 *
 *  FUNCTIONS
 *  CALLED:        query()            screen.c
 *                 clear_screen()  screen.c
 *                 get_reg()       stackops.c
 *                 disp_stack()    stackops.c
 *                 enter_reg()     stackops.c
 *                 clear_stack()   stackops.c
 *                 stack()         stack.c
 *                 fir_setup()     firsetup.c
 *
 *
 *  AUTHOR:        Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include "rolf.h"
#include "fir.h"

/* template defined in fir.h, this object contains the
   parameters which define the filter response
   characteristics                                       */
extern FIR_SPECS fir;

extern double rho; /* estimate of the Chebyshev error    */

extern grid_density[], /* this array allows a choice of
                          frequency grid densities       */
       grid_choice;    /* index to the current choice    */


int fircoef()
{
    REG *x; /* pointer to a ROLF register                */

    x = get_reg(X); /* point it to the X register        */

    FOREVER
        {
        clear_screen();
        disp_stack();
        printf("\n0:  Exit");
        printf("\n1:  Compute via Parks-Mclellan");
        printf("\n2:  Compute New Method");
        printf("\n3:  Toggle frequency grid density: (%d)",
            grid_density[grid_choice]);
        printf("\n    <RETURN> Stack");
        printf("\n\n>>  ");
        switch (query("") - '0')
            {
            case EXIT:
                return(OK);

            case 1:
/*----------------------------------------------------------*/
/*              Duplicate the X register to Y, load it     */
/*              with the appropriate information and       */
/*              invoke the setup function which will then */
/*              then invoke the Remez exchange             */
/*----------------------------------------------------------*/
                enter_reg();
                clear_screen();
                x->len = fir.order;
                x->type = REAL;
                x->contents = FIR_COEFFICIENTS;
                   if (fir_setup(x->reg, &rho,
                       PARKS_McCLELLAN))
                    query("Error computing coefficients");
                   break;

            case 2:
/*----------------------------------------------------------*/
/*              Duplicate the X register to Y, load it     */
/*              with the appropriate information and       */
/*              invoke the setup function which will then */
/*              invoke the new method                      */
/*----------------------------------------------------------*/
                enter_reg();
                  clear_screen();
                x->len = fir.order;
                x->type = REAL;
                x->contents = FIR_COEFFICIENTS;
                   if (fir_setup(x->reg, &rho, NEW_METHOD))
                    query("Error computing coefficients");
                break;

            case 3:
/*------------- roll the frequency grid density ----------*/
                STACKROLL(grid_density, grid_choice, EOF);
                continue;

            default:
/*------------- stack register manipulations function ----*/
                if (stack())
                    query("Error in stack ops");
                continue;
            } /* end inner switch */
        } /* end FOREVER */
} /* end fir_coef() */
```

```
/**********************************************************************
 *
 *
 *  SOURCE FILE:    firsetup.c
 *
 *
 *  FUNCTION:       int fir_setup(res, rho, method)
 *
 *
 *  ARGUMENTS:      (input/output) COMPLEX res[]
 *                      The filter impulse coefficients are
 *                  returned in the real part of this array.
 *                  If the approximation was to a general
 *                  response type, then this desired arbi-
 *                  trary magnitude response would be passed
 *                  into the function from the X register.
 *
 *                  (output) double * rho
 *                      This pointer references the final
 *                  estimate of the Chebyshev error upon
 *                  exiting the function.
 *
 *                  (input) int method
 *                      Specifies whether the new method
 *                  or the Remez exchange will be invoked
 *                  to obtain the filter coefficients.
 *
 *
 *  RETURN:         int: OK
 *                       ERROR
 *
 *
 *  DESCRIPTION:
 *
 *      This function determines if the filter is of
 *      case 1, 2, 3, or 4, determines the grid length
 *      and the number of approximating cosines.  It
 *      then invokes the Remez exchange if an optimal
 *      solution is desired.  For the new method, it
 *      estimates the Chebyshev error based on a
 *      guessed set of equally-spaced extremals and
 *      sets up to do the Lagrange interpolation.
 *      Finally, it obtains the filter coefficients
 *      by performing a DCT on the filter magnitude
 *      response.
 *
 *      case 1: odd length / even symmetry
 *
 *      case 2: even length / even symmetry
 *
 *      case 3: odd length / odd symmetry
 *
 *      case 4: even length / odd symmetry
 *
 *
 *  DOCUMENTATION
 *  FILES:          A NEW METHOD FOR THE DESIGN
 *                  OF FIR DIGITAL FILTERS
 *
 *
 *  CONSTANTS:      OK                  common.h
 *                  ERROR               common.h
 *                  MAXLEN              common.h
 *                  PARKS_McCLELLAN     fir.h
 *                  NEW_METHOD          fir.h
 *                  EVEN_SYMMETRY       fir.h
 *                  ODD_SYMMETRY        fir.h
 *                  M_2PI               smath.h
 *                  M_PI                smath.h
 *                  M_SQRT2             smath.h
 *
 *
 *  MACROS
 *  EXPANDED:       ODD()   smath.h
 *
 *
 *  FUNCTIONS
 *  CALLED:         make_res()               makeres.c
 *                  query()                  screen.c
 *                  estimate_extremals()     estextr.c
 *                  estimate_rho()           estrho.c
 *                  lagrange()               lagrange.c
 *                  dct()                    dct.c
 *                  alphas()                 alphas.c
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include "common.h"
#include "smath.h"
#include "fir.h"

#define DCT   1  /* forward discrete cosine transform     */

/* template in fir.h, this object contains the indexes
   into the dense grid for the transition frequencies     */
*/
extern TRAN_INDEXES tran_index[];


/* template in fir.h,  this object contains the filter
   parameters which specify all its response character-
   istics                                                  */
extern FIR_SPECS fir;

/* adjustable density for the frequency grid               */
int grid_density[] = {5,16,20,25,EOF},
                     grid_choice = 1; /* initialize to 16 */

int fir_setup(COMPLEX res[], double *rho, int method)
{
    int     i,
            initial,   /* starting index of cosines, error,
                          and frequency grids              */
            final,     /* ending index of the same grids   */

            grid_len,  /* the number of frequencies in the
                          dense grid excluding those which
                          fall in the transition regions   */

            n,         /* number of approximating cosines */

            no_points, /* number of frequencies in the
                          dense grid including the tran-
                          sition regions                   */

            *new_pntr, /* updated array of extremal
                          indexes into the cosine, error,
                          and frequency grids              */

            *old_pntr, /* previous array of extremal
                          indexes into the cosine, error,
                          and frequency grids              */

            fil_type;  /* filter type: cases 1, 2, 3, 4   */

    double  f,         /* temporary frequency variable     */
```

```
           delta_f,    /* frequency spacing of dense grid */

           *a, *b,     /* arrays used in the Lagrange     */
           *c, *x,     /* interpolation  function         */

           *err,       /* error function over which the
                          extremal frequencies are searched
                          for                             */

           *grid,      /* dense grid of frequencies
                          excluding transition regions    */

           *cosines,   /* cosines of the dense freq. grid */

           factor,     /* used to transform magnitude
                          response before usng the DCT
                          to get coefs.                   */

           sum;        /* used in transforming magnitude
                          response before using the DCT
                          to get coefficients             */


/*----------------------------------------------------------*/
/*  determine filter type and number of approximating       */
/*  cosines                                                 */
/*----------------------------------------------------------*/
    if (fir.type == EVEN_SYMMETRY)
        {
        if (ODD(fir.order))
            {
            fil_type = 1;
            n = (fir.order - 1) / 2 + 1;
            }
        else
            {
            fil_type = 2;
            n = fir.order / 2;
            }
        }
    else
    if (fir.type == ODD_SYMMETRY)
        {
        if (ODD(fir.order))
            {
            fil_type = 3;
            n = (fir.order - 1) / 2;
            }
        else
            {
```

```
            fil_type = 4;
            n = fir.order / 2;
            }
        }

/*-----------------------------------------------------------*/
/*  determine the number of frequencies in the dense grid */
/*  including those which fall in the transition regions  */
/*-----------------------------------------------------------*/
    no_points = grid_density[grid_choice] * n;
    if (no_points > MAXLEN)
        {
        query("Error: frequency grid too long ");
        return(ERROR);
        }


/*-----------------------------------------------------------*/
/*  Allocate the storage for all the working arrays, note */
/*  that the storage for new_pntr is twice that of        */
/*  old_pntr.  This  is for the case in which more than   */
/*  (n+1) local maximums or minimums are found when       */
/*  searching the error function.                         */
/*-----------------------------------------------------------*/
    old_pntr = (int *) malloc(sizeof(int) * (3*n + 3));
    if (old_pntr == NULL)
        {
        query("Error:'old_pntr' not malloced"
            " in fir_filter() ");
        return(ERROR);
        }

    new_pntr = (int *) (old_pntr + n + 1);

    x = (double *) malloc(sizeof(double) *
        (n*4+2 + 3*no_points+3));

    if (x == NULL)
        {
        free(old_pntr);
        query("Error:'x' not mallocated in fir_filter() ");
        return(ERROR);
        }

    a = (double *) (x + n + 1);
    b = (double *) (a + n + 1);
    c = (double *) (b + n);
    err     = (double *) (c + n);
    grid    = (double *) (err + no_points + 1);
    cosines = (double *) (grid + no_points + 1);
```

```
/*- determing the spacing of the dense set of frequencies */
    delta_f = 0.5 / (no_points - 1);

/*------------------------------------------------------*/
/*  generate the magnitude response for which the         */
/*  Chebyshev approximation will be fitted to             */
/*------------------------------------------------------*/
    if (make_res(res, grid, cosines, delta_f,
        no_points, &grid_len))
        {
        free(old_pntr);
        free(x);
           query("Error: fir_filter() ");
        return(ERROR);
        }

/*------------------------------------------------------*/
/*  determine the starting and ending grid indexes based  */
/*  on the filter type and make the necessary adjustments */
/*  in the  desired response and weighting for filters    */
/*  other than case 1, i.e., odd length and even symmetry */
/*------------------------------------------------------*/
    switch (fil_type)
        {
        case 1:
            initial = 0;
            final = grid_len;
            break;

        case 2:
            initial = 0;
            final = grid_len - 1;
            for (i = 0; i <= grid_len-1; ++i)
                {
                res[i].re = res[i].re / cos(M_PI * grid[i]);
                res[i].im = res[i].im * cos(M_PI * grid[i]);
                }
            break;

        case 3:
            initial = 1;
            final = grid_len - 1;
            for (i = 1; i <= grid_len-1; ++i)
                {
                res[i].re = res[i].re / sin(M_2PI *grid[i]);
                res[i].im = res[i].im * sin(M_2PI *grid[i]);
                }
            break;
        case 4:
            initial = 1;
            final = grid_len;
            for (i = 1; i <= grid_len; ++i)
                {
                res[i].re = res[i].re / sin(M_PI * grid[i]);
                res[i].im = res[i].im * sin(M_PI * grid[i]);
                }
            break;
        } /* end of switch */

/*------------------------------------------------------------*/
/*  make an initial equally-spaced guess of the extremal     */
/*  frequencies                                               */
/*------------------------------------------------------------*/
    if (estimate_extremals(initial, final, n,
        new_pntr, grid_len))
        {
        free(x);
        free(old_pntr);
        query("Error: fir_filter() ");
        return(ERROR);
        }

/*------------------------------------------------------------*/
/*  if the Park's method is desired, than do the Remez       */
/*  exchange                                                  */
/*------------------------------------------------------------*/
    if (method == PARKS_McCLELLAN)
        {
        if (remez(res, cosines, err, a, b, c, x, rho,
            initial, final, old_pntr, new_pntr, n))
            {
            free(x);
            free(old_pntr);
            query("Error: fir_filter() ");
            return(ERROR);
            }
        }
    else
/*------------------------------------------------------------*/
/*  for the new method, compute a,b,c,x for the Lagrange     */
/*  interpolation formula                                     */
/*------------------------------------------------------------*/
    if (method == NEW_METHOD)
        *rho = estimate_rho(res, cosines, n, new_pntr,
            a, b, c, x);
    else
        {
        free(x);
        free(old_pntr);
        query("Error: fir_filter() ");
        return(ERROR);
        }

/*--------------------------------------------------------*/
/*  Do the transformation on the desired magnitude        */
/*  response  prior to performing a DCT. The development  */
/*  of this transformation is the primary reason for the  */
/*  existence of this thesis.                             */
/*--------------------------------------------------------*/
    sum = 0;
    for (i = 0; i <= n-1; ++i)
        {
        f = (2.0*i+1.0)/(4.0*n);
        res[i].re = lagrange(cos(M_2PI * f), b, c, x, n);
        res[i].im = 0;
        sum += res[i].re;
        }

    factor = -(M_SQRT2 - 1) * sum / n / M_SQRT2;
    for (i = 0; i <= n-1; ++i)
        res[i].re = (res[i].re + factor) * sqrt(2.0 / n);

    free(x);        /* these are no longer needed         */
    free(old_pntr);

/*- do the DCT - this yields the cosine coefficients -----*/
    if (dct(res, n, DCT))
        {
        query("Error: fir_filter() ");
        return(ERROR);
        }

/*--------------------------------------------------------*/
/*  convert the approximating cosine coefficients to      */
/*  impulse  coefficients                                 */
/*--------------------------------------------------------*/
    if (alphas(res, fil_type, fir.order, n))
        {
        query("Error: fir_filter() ");
        return(ERROR);
        }

    return(OK);

} /* end fir_setup() */
```

```
/*************************************************************
 *
 *
 *  SOURCE FILE:   remez.c
 *
 *
 *  FUNCTION:      int remez(res, cosines, err, a, b, c, x,
 *                           new_rho, initial, final,
 *                           old_pntr, new_pntr, n)
 *
 *
 *  ARGUMENTS:
 *
 *      (input/output) COMPLEX res[]
 *          Passes the desired filter response in the
 *      real part  and the weighting function in the
 *      imaginary part, then returns the filter
 *      impulse coefficients in the real part.
 *
 *      (input) double cosines[]
 *          This array contains the cosines of the
 *      dense grid of frequencies.
 *
 *      double err[]
 *          Used within the remez() fnc. to store the
 *      error function over which the extremal frequencies
 *      are searched for.
 *
 *      (input) double a[], b[], c[], x[]
 *          Arrays used by the Lagrange interpolation
 *      function.
 *
 *      (output) double *new_rho
 *          The final value of the Chebyshev error is
 *      referenced by this pointer.
 *
 *      (input) int initial, final
 *          These integers are the first and last
 *      indexes into the dense grid of frequencies and
 *      are set according to the filter type, i.e.,
 *      even or odd symmetry and even or odd impulse
 *      length. initial is set to zero or one and final
 *      is set to the frequency grid length or frequency
 *      grid length minus one. This adjustment prevents
 *      a divide by zero when normalizing the magnitude
 *      response and weighting function so that the
 *      Remez exchange can be used for all four filter
 *      types.
 *
 *      (input) int new_pntr[]
*           The initial guessed set of extremal freq-
*       uency indexes is passed and thereafter, it
*       contains the current updated set of extremal
*       indexes.
*
*       int old_pntr[]
*           This array contains the previous set
*       of extremal frequency indexes and is used for
*       comparison purposes to determine if any
*       changes occured.
*
*       (input) int n
*           This is the number of approximating
*       cosines.
*
*
*   RETURN:         int: OK
*                        ERROR
*
*
*   DESCRIPTION:
*
*       Performs the remez exchange.  This involves
*       searching over a dense set of frequencies
*       for those at which the weighted error function
*       achieves its maxima or minima.  These new
*       extremal frequencies are then used in the
*       Lagrange interpolation routine to generate a
*       new approximation to the desired response.
*       A new weighted error function is then computed
*       and the dense grid of frequencies is once again
*       searched for those at which the error function
*       changes sign.  This is continued until no
*       extremal frequencies have changed location
*       from the previous iteration.  The Remez
*       exchange is then said to have converged and
*       the approximation problem has been solved.
*
*
*   DOCUMENTATION
*   FILES:          None
*
*
*   CONSTANTS:      OK        common.h
*                   ERROR     common.h
*                   MAX_ITER  fir.h
*
*   MACROS
*   EXPANDED:       None
*
```

```
 *
 *  FUNCTIONS
 *  CALLED:         estimate_rho()      estrho.c
 *                  query()             screen.c
 *                  lagrange()          lagrange.c
 *                  find_extremals()    findextr.c
 *                  choose_endpoints()  choosend.c
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 ************************************************************/

#include <stdio.h>
#include <math.h>
#include "common.h"
#include "screen.h"
#include "smath.h"
#include "fir.h"

/* index into the dense grid of frequencies specifying
   which are the transition frequencies                    */
extern TRAN_INDEXES tran_index[];


int remez(COMPLEX res[], double cosines[], double err[],
          double a[], double b[], double c[], double x[],
          double *new_rho, int initial, int final,
          int old_pntr[], int new_pntr[], int n)
{
    int     i,
            iter, /* counter for iterations of the Remez
                     exchange                              */

            count, /* the number of extremals that were
                      found                                */

            changes; /* the number of extremals that
                        changed                            */

    double  old_rho; /* the Chebyshev error of the previous
                        iteration                          */
    *new_rho = 0;
    iter = 1;
    do
        {
        old_rho = *new_rho;

/*----------------------------------------------------------*/
/*      estimate the error and compute the  vectors         */
/*      to be used by the Lagrange interpolation fnc.       */
/*----------------------------------------------------------*/
        *new_rho = estimate_rho(res, cosines, n, new_pntr,
            a, b, c, x);

/*----- make sure the error is converging to a maximum ---*/
        if (old_rho > *new_rho)
            {
            query("Error: Deviation did not increase"
                " => suboptimal ");
            break;
            }

/*----- compute the error function ----------------------*/
        for (i = initial; i <= final; ++i)
            err[i] = res[i].im * (res[i].re -
                lagrange(cosines[i], b, c, x, n));

/*----------------------------------------------------------*/
/*      Search the error function for all the maxima and    */
/*      minima without regard for the number found or the */
/*      Alternation Theorem.                                */
/*----------------------------------------------------------*/
        if (find_extremals(err, new_pntr,
            *new_rho, initial, final, &count))
            {
            query("Error: remez() ");
            return(ERROR);
            }

/*----------------------------------------------------------*/
/*      Now make sure that the right number of extremal     */
/*      frequencies were found and that they all satisfy    */
/*      the Alternation Theorem.                            */
/*----------------------------------------------------------*/
        if (choose_endpoints(err, new_pntr, old_pntr,
            n, count,
            &changes))
            {
            query("Error: remez() ");
            return(ERROR);
            }
/*----- repeat until no extremals changed or maximum iter */
        } while ((++iter <= MAX_ITER) && (changes != 0));

    return(OK);

} /* end remez() */
```

```
/************************************************************
 *
 *
 *  SOURCE FILE:   makeres.c
 *
 *
 *  FUNCTION:      int make_res(res, grid, cosines, delta_f,
 *                              no_points, grid_len)
 *
 *
 *  ARGUMENTS:
 *
 *      (output) COMPLEX res[]
 *          Returns the desired GENERAL or BRICKWALL
 *      filter magnitude response in the real part and
 *      the weighting function in the imaginary part.
 *      The use of a complex array is not special but
 *      adds convenience later when returning the
 *      impulse coefficients.  This complex array
 *      can then be FFT'd to yield the simulated
 *      magnitude response of the filter.  The magnitude
 *      and weighting function for each band of the
 *      filter are derived from the FIR filter parameter
 *      object named fir.  If the filter magnitude
 *      response is GENERAL (arbitrary), then the
 *      weighting function is set to unity throughout.
 *
 *      (input) double grid[]
 *          This is the dense grid of frequencies.
 *
 *      (input) double cosines[]
 *          This array contains the cosines of the
 *      dense grid of frequencies.
 *
 *      (input) double delta_f
 *          Spacing between the frequencies in the
 *      dense grid.
 *
 *      (intput) int no_points
 *          The number of approximating cosines times
 *      the frequency grid density.
 *
 *      (output) int * grid_len
 *          The number of dense grid frequencies.
 *      This is equal to no_points minus the number
 *      frequencies that fall within the transition
 *      regions.
 *
 *
 *  RETURN:        int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      Generates the desired GENERAL or BRICKWALL
 *      magnitude response and weighting function
 *      to be passed to the Remez exchange for fitting
 *      an approximation to the desired response.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None
 *
 *
 *  CONSTANTS:      OK          common.h
 *                  M_2PI       smath.h
 *                  BRICKWALL   fir.h
 *                  GENERAL     fir.h
 *
 *
 *  MACROS
 *  EXPANDED:       ROUND()  smath.h
 *
 *
 *  FUNCTIONS
 *  CALLED:         None
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 ****************************************************************/

#include <stdio.h>
#include <math.h>
#include "common.h"
#include "smath.h"
#include "fir.h"

/* template in fir.h, index into the dense grid of
   frequencies specifying which are the transition
   frequencies                                               */
extern TRAN_INDEXES tran_index[];
extern FIR_SPECS fir; /* template in fir.h,  this object
                         contains the filter parameters
                         which specify its characteristics*/


int make_res(res, grid, cosines, delta_f,
    no_points, grid_len)
    COMPLEX res[];
    double  grid[], cosines[], delta_f;
    int     no_points, *grid_len;
{
    int     i, j, k;

    double  f; /* temporary frequency variable            */

    switch (fir.response)
        {
        case BRICKWALL:
            tran_index[0].lw = 0;
            for (i = 0; i < fir.nobands-1; ++i)
                {
                k = ROUND((fir.tran_freq[i].up -
                    fir.tran_freq[i].lw) / delta_f);
                tran_index[i].up = tran_index[i].lw + k;
                tran_index[i+1].lw = tran_index[i].up + 1;
                }
            k = ROUND((fir.tran_freq[i].up -
                fir.tran_freq[i].lw) / delta_f);
            tran_index[i].up = tran_index[i].lw + k;
            *grid_len = tran_index[i].up;

            f = 0;
            for (i = 0; i < fir.nobands; ++i)
                {
                f = fir.tran_freq[i].lw;
                for (j = tran_index[i].lw;
                    j < tran_index[i].up; ++j)
                    {
                    res[j].re = fir.band_value[i];
                    res[j].im = 1/fir.band_weight[i];
                    grid[j] = f;
                    cosines[j] = cos(M_2PI * f);
                    f += delta_f;
                    }
                f = fir.tran_freq[i].up;
                res[j].re = fir.band_value[i];
                res[j].im = 1/fir.band_weight[i];
                grid[j] = f;
                cosines[j] = cos(M_2PI * f);
                }

            break;

        case GENERAL:
/* NOT AVAILABLE YET !
            f = 0;
            *grid_len = no_points-1;
            fir.nobands = 1;
            tran_index[0].lw = 0;
            tran_index[0].up = *grid_len;

            for (i = 0; i <= *grid_len; ++i)
                {
                res[i].im = 1.0;
                grid[i] = f;
                cosines[i] = cos(M_2PI * f);
                f += delta_f;
                }
*/
            break;
        } /* end switch */

    return(OK);

} /* end make_response() */
```

```
/***************************************************************
 *
 *
 *  SOURCE FILE:   estextr.c
 *
 *
 *  FUNCTION:      int estimate_extremals(initial, final,
 *                                       n, new_pntr,
 *                                       grid_len)
 *
 *
 *  ARGUMENTS:
 *
 *      (input) int initial, final
 *          These integers are the first and last
 *      indexes into the dense grid of frequencies and
 *      are set according to the filter type, i.e.,
 *      even or odd symmetry and even or odd impulse
 *      length. initial is set to zero or one and final
 *      is set to the frequency grid length or frequency
 *      grid length minus one. This adjustment prevents
 *      a divide by zero when normalizing the magnitude
 *      response and weighting function so that the
 *      Remez exchange can be used for all four filter
 *      types.
 *
 *      (input) int n
 *          This is the number of approximating
 *      cosines.
 *
 *      (output) int new_pntr[]
 *          Indexes of the  initial set of extremal
 *      frequencies are guessed and returned.
 *
 *      (input) int grid_len
 *          The number of dense grid frequencies.
 *      This is equal to the number of approximating
 *      cosines times the grid density minus the
 *      number frequencies that fall within the
 *      transition regions.
 *
 *
 *  RETURN:        int: OK
 *                      ERROR
 *
 *
 *  DESCRIPTION:
 *
 *      Makes a stab at guessing the initial set
 *      of extremals of the error function. These
 *      are equally spaced within regions but not
 *      within the overall frequency grid which
 *      includes the transition frequencies.
 *
 *
 *  DOCUMENTATION
 *  FILES:         None
 *
 *
 *  CONSTANTS:     OK     common.h
 *                 ERROR  common.h
 *
 *
 *  MACROS
 *  EXPANDED:      ROUND()  smath.h
 *
 *
 *  FUNCTIONS
 *  CALLED:        None
 *
 *
 *  AUTHOR:        Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 *************************************************************/

#include <stdio.h>
#include <math.h>
#include "common.h"
#include "smath.h"
#include "fir.h"


/* template in fir.h, index into the dense grid of
   frequencies specifying which are the transition
   frequencies                                             */
extern TRAN_INDEXES tran_index[];


extern FIR_SPECS fir; /* template in fir.h,  this object
                         contains the filter parameters    */


int estimate_extremals(initial, final, n, new_pntr,
    grid_len)
    int initial, final, n, grid_len, new_pntr[];
{
    int     i, j, k,
            count = 0;

    double  f, findex, fdelta;

    for (i = 0; i < fir.nobands; ++i)
        {
        if (i == fir.nobands-1)
            k = n - count;
        else
            {
            f = (n+1) * (tran_index[i].up -
                tran_index[i].lw + 1) /(grid_len+1)-1;
            k = (int)ceil(f);

            if (k <= 0)
                {
                query("Error: transition width too narrow");
                return(ERROR);
                }
            }

        fdelta = (tran_index[i].up - tran_index[i].lw) / k;
        findex = (double) tran_index[i].lw;

        for (j = 0; j <= k-1; ++j)
            {
            new_pntr[count] = ROUND(findex);
            findex += fdelta;
            ++count;
            }

        new_pntr[count] = tran_index[i].up;
        ++count;
        }
    new_pntr[0] = initial;
    new_pntr[n] = final;

    return(OK);

} /* end estimate_extremals() */
```

```
/*********************************************************************
 *
 *
 *  SOURCE FILE:   findextr.c
 *
 *
 *  FUNCTION:      int find_extremals(err, new_pntr, rho,
 *                                initial, final, count)
 *
 *
 *  ARGUMENTS:
 *
 *      (input) double err[]
 *          The error function over which the
 *      maxima and minima are searched for.
 *
 *      (input) int new_pntr[]
 *          Indexes of the current updated extremal
 *      frequencies in the dense grid.
 *
 *      (input) double rho
 *          An estimate of the Chebyshev error.
 *
 *      (input) int initial, final
 *          These integers are the first and last
 *      indexes into the dense grid of frequencies and
 *      are set according to the filter type, i.e.,
 *      even or odd symmetry and even or odd impulse
 *      length. initial is set to zero or one and final
 *      is set to the frequency grid length or frequency
 *      grid length minus one. This adjustment prevents
 *      a divide by zero when normalizing the magnitude
 *      response and weighting function so that the
 *      Remez exchange can be used for all four filter
 *      types.
 *
 *      (output) int * count
 *          The total number of extremal frequencies
 *      that were found.
 *
 *
 *  RETURN:        int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      This routine finds every local maximum or
 *      minimum of the error curve.  It makes no
 *      attempt to choose based on the Alternation
 *      Theorem or upon the number (n+1) extremals
 *      that will eventually be retained.  Those
 *      discriminations will be performed in the
 *      function choose_endpoints().
 *
 *
 *  DOCUMENTATION
 *  FILES:         None
 *
 *
 *  CONSTANTS:     OK  common.h
 *
 *
 *  MACROS
 *  EXPANDED:      SIGN()  smath.h
 *
 *
 *  FUNCTIONS
 *  CALLED:        None
 *
 *
 *  AUTHOR:        Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include "common.h"
#include "smath.h"
#include "fir.h"


/* template in fir.h, this array is the indexes into the
   dense grid of frequencies specifying which are the
   transition frequencies                                 */
extern TRAN_INDEXES tran_index[];


int find_extremals(double err[], int new_pntr[], double rho,
    int initial, int final, int *count)
{
    int i, /* used to index thru the error function when
              searching for local maxima/minima           */

        oldsign,  /* for comparing to determine when a
        newsign,     change of sign has occured which
                     indicates a max or min was passed    */

        tran_pntr; /* indexes through the distinct bands  */

/*- provide a safty margin from roundoff -----------------*/
    rho = rho * 0.999;

/*- initialize for finding the first extremal ------------*/
    oldsign = SIGN(err[initial+1] - err[initial]);

    tran_pntr = 0; /* start with first (or only) band     */

/*- assume the first freq. in grid is always an extremal -*/
    new_pntr[0] = initial;
    *count = 1;  /* at least one extremal will be found   */

    i = initial+1; /* next frequency                      */

    while (i < final) /* final indexes the last freq.     */
        {
/*------------------------------------------------------*/
/*      make sure that all transition frequencies are    */
/*      chosen as extremals regardless if they are       */
/*      maxima or minima                                 */
/*------------------------------------------------------*/
        if (i == tran_index[tran_pntr].up)
            {
            new_pntr[*count] = tran_index[tran_pntr].up;
            ++*count;
            ++i;
            ++tran_pntr;
            new_pntr[*count] = tran_index[tran_pntr].lw;
            ++*count;
            oldsign = SIGN(err[i+1] - err[i]);
            ++i;
            }
        else
/*------------------------------------------------------*/
/*          if not a transition frequency, then search for*/
/*          a local maximum or minimum                   */
/*------------------------------------------------------*/
            {
            newsign = SIGN(err[i+1]-err[i]);

/*------------------------------------------------------*/
/*         When a local max/min is found, snatch it if   */
/*         its magnitude is greater than the absolute    */
/*         value of the error function at that frequency */
/*------------------------------------------------------*/
```

```
            if ((newsign != oldsign) &&
                (fabs(err[i]) > rho))
                {
                new_pntr[*count] = i;
                ++*count;
                }

            oldsign = newsign; /* update for next extremal*/
            ++i;
            }
        }

/*- choose the last frequency to be an extremal also -----*/
    new_pntr[*count] = final;
    return(OK);

} /* end find_extremals() */
```

```
/***********************************************************
 *
 *
 *  SOURCE FILE:    choosend.c
 *
 *
 *  FUNCTION:       choose_endpoints(err, new_pntr, old_pntr,
 *                                   n, count, changes)
 *
 *
 *  ARGUMENTS:
 *
 *      (input) double err[]
 *          This contains the error function over
 *      which the extremal frequencies are searched
 *      for.
 *
 *      (input) int new_pntr[]
 *          The indexes into the dense grid of the
 *      updated extremal frequencies.
 *
 *
 *      int old_pntr[]
 *          This array contains the previous set
 *      of extremal frequency indexes and is used for
 *      comparison purposes to determine if any
 *      changes occured. If none did, then the Remez
 *      exchange is completed.
 *
 *      (input) int n
 *          This is the number of approximating
 *      cosines.
 *
 *      (input) int count
 *          The total number of extremal frequencies
 *      found in find_extremals().
 *
 *      (output) int * changes
 *          The number of extremals that changed
 *      between the last and next to last iteration.
 *
 *
 *  RETURN:         int: OK
 *
 *
 *  DESCRIPTION:
 *
 *      Inspects the end-point extremals and makes
 *      certain that all extremals satisfy the
 *      Alternation Theorem.  If two adjacent
 *      extremal frequencies are found which
 *      correspond to error values of the same
 *      sign, then the extremal corresponding to
 *      the smallest absolute value of the error
 *      function is discarded. Then, if the excess
 *      number of extremals found is divisible by two,
 *      i.e., n+3 or more,  then the two extremals
 *      corresponding to the smallest errors are discarded
 *      jointly.  This is necessary so that the sign
 *      convention of the Alternation Theorem will not
 *      be violated.  This process is continued until only
 *      n+1 or n+2 extremals remain.  If it is n+2, then the
 *      end points are inspected for the largest absolute
 *      value of the error function.  The other end point
 *      extremal frequency is discarded.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None
 *
 *
 *  CONSTANTS:      OK  common.h
 *
 *
 *  MACROS
 *  EXPANDED:       SIGN()  smath.h
 *
 *
 *  FUNCTIONS       None
 *  CALLED:
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 *****************************************************************/

#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include "common.h"
#include "smath.h"
#include "fir.h"
```

```
int choose_endpoints(err, new_pntr, old_pntr,
    n, count, changes)
    int    new_pntr[], old_pntr[], n, count, *changes;
    double err[];
{
    int i, j, index;
    double temp;

    i = 1;
/*---------------------------------------------------------*/
/*  Check every extremal found to make sure that its       */
/*  corresponding error function value alternates in sign */
/*  with its neighbors, if it doesn't then discard the     */
/*  smallest of the group                                  */
/*---------------------------------------------------------*/
    while (i <= count)
        {
        if (SIGN(err[new_pntr[i]]) ==
            SIGN(err[new_pntr[i-1]]))
            {
            if (fabs(err[new_pntr[i]]) >
                fabs(err[new_pntr[i-1]]))
                {
                for (j = i; j <= count; ++j)
                    new_pntr[j-1] = new_pntr[j];
                }
            else
                {
                for (j = i; j <= count-1; ++j)
                    new_pntr[j] = new_pntr[j+1];
                --count;
                }
            }
        else
            ++i;
        } /* end of while */

/*---------------------------------------------------------*/
/*  If all extremals satisify the alternation theorem but */
/*  more than (n+2) exist, then discriminating the end-   */
/*  points will not suffice and the smallest extremal     */
/*  with its corresponding smallest neighbor must be      */
/*  discarded.  The alternation theorm is then still      */
/*  satisified                                            */
/*---------------------------------------------------------*/
    while (count >= n+2)
        {
        temp = 1e30; /* I hope this is big enough !       */

        for (i = 1; i <= count-1; ++i)
```

```
            {
            if (fabs(err[new_pntr[i]]) < temp)
                {
                temp = fabs(err[new_pntr[i]]);
                index = i;
                }
            }

        if (fabs(err[new_pntr[index-1]]) <
            fabs(err[new_pntr[index+1]]))
             --index;

        for (j = index; j <= count-2; ++j)
            new_pntr[j] = new_pntr[j+2];
        count -= 2;
        } /* end of while */

/*-----------------------------------------------------------*/
/* If (n+2) extremals exist which satisify the               */
/* alternation  theorem, then choose the frequency           */
/* with the corresponding error of largest absolute          */
/* value                                                     */
/*-----------------------------------------------------------*/
    if (count == n+1)
        {
        if (fabs(err[new_pntr[count]]) >
            fabs(err[new_pntr[0]]))

            for (i = 0; i <= count-1; ++i)
                new_pntr[i] = new_pntr[i+1];

        }

/*-----------------------------------------------------------*/
/*  see how many extremals have changed from the previous */
/*  iteration of the Remez exchange                          */
/*-----------------------------------------------------------*/
    *changes = 0;

    for (i = 0; i <= n; ++i)
        {
        if (old_pntr[i] != new_pntr[i])
            ++*changes;

        old_pntr[i] = new_pntr[i];
        }

    return(OK);

} /* end choose_endpoints() */
```

```
/*****************************************************************
 *
 *
 *  SOURCE FILE:   estrho.c
 *
 *
 *  FUNCTION:      double estimate_rho(res, cosines, n,
 *                                     new_pntr, a, b, c, x)
 *
 *
 *  ARGUMENTS:
 *
 *      (input) COMPLEX res[]
 *          The real part of res[] contains the desired
 *      magnitude response of the filter and the imaginary
 *      part contains the weighting function.
 *
 *      (input) double cosines[]
 *          This array contains the cosines of the
 *      dense grid of frequencies.
 *
 *      (input) int n
 *          This is the number of approximating
 *      cosines.
 *
 *      (input) int new_pntr[]
 *          Indexes of the current updated set of
 *      extremal frequencies in the dense grid.
 *
 *      (output) double a[], b[], c[], x[]
 *          Vectors used by the lagrange interpolation
 *      function.
 *
 *
 *  RETURN:        double: Value represents an estimate of
 *                         the Chebyshev error for a given
 *                         set of extremal frequencies.
 *
 *
 *  DESCRIPTION:
 *
 *      Estimates the Chebyshev error for a given set
 *      of extremal frequencies and computes the
 *      arrays a,b,c,x which will be used by the
 *      Lagrange interpolation function.
 *
 *
 *  DOCUMENTATION
 *  FILES:         None
 *
 *
 *  CONSTANTS:      None
 *
 *
 *  MACROS
 *  EXPANDED:       None
 *
 *
 *  FUNCTIONS
 *  CALLED:         None
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include <math.h>
#include "common.h"
#include "smath.h"

double estimate_rho(res, cosines, n, new_pntr, a, b, c, x)
    int     n, new_pntr[];
    double  cosines[], a[], b[], c[], x[];
    COMPLEX res[];
{
    int     i, j;
    double  num,
            den,
            rho;

/*- compute the cosines of the extremal frequencies ------*/
    for (i = 0; i <= n; ++i)
        x[i] = cosines[new_pntr[i]];

    for (i = 0; i <= n; ++i)
        {
        a[i] = 1;
        for (j = 0; j <= n; ++j)
            if (i != j)
                {
                a[i] = a[i] / (x[i] - x[j]);
                if (i < n && j < n)
```

```
                        b[i] = a[i];
                    }
            }

        num = 0;
        den = 0;
        for (i = 0; i <= n; ++i)
            {
            num += a[i] * res[new_pntr[i]].re;
            den += pow(-1.0,(double)i) * a[i] /
            res[new_pntr[i]].im;
            }

        rho = num / den;
        for (i = 0; i <= n-1; ++i)
            c[i] = res[new_pntr[i]].re -
            pow(-1.0,(double)i) * rho / res[new_pntr[i]].im;

        return(fabs(rho));

    } /* end estimate_rho() */
```

```
/*****************************************************************
 *
 *
 *  SOURCE FILE:    lagrange.c
 *
 *
 *  FUNCTION:       double lagrange(cosf, b, c, x, n)
 *
 *
 *  ARGUMENTS:
 *
 *      (input) double cosf
 *          The cosine value of the frequency of
 *      interest.
 *
 *      (input) double  b[], c[], x[]
 *          Arrays generated by calling the fnc.
 *      estimate_rho().
 *
 *      (input) int n
 *          The number of approximating cosines.
 *
 *
 *  RETURN:         double: Value represents an estimate for
 *                          the function at the point of
 *                          interest.
 *
 *
 *  DESCRIPTION:    Performs the Lagrange interpolation in
 *                  the barycentric form.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None
 *
 *
 *  CONSTANTS:      None
 *
 *
 *  MACROS
 *  EXPANDED:       None
 *
 *
 *  FUNCTIONS
 *  CALLED:         None
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 **********************************************************/

#include <stdio.h>
#include "common.h"

double lagrange(double cosf, double b[], double c[],
                double x[], int n)
{
    int i;

    double temp,
           num, den;

/*----------------------------------------------------------*/
/*  if cosf equals any of the x[i], then return c[i] to    */
/*  prevent a divide by zero                                */
/*----------------------------------------------------------*/
    for (i = 0; i <= n-1; ++i)
        if (cosf == x[i])
            return(c[i]);

/*- initialize and perform the interpolation -------------*/
    num = 0;
    den = 0;

    for (i = 0; i <= n-1; ++i)
        {
        temp = b[i] / (cosf - x[i]);
        num += temp * c[i];
        den += temp;
        }

    return(num / den);
}
```

```
/***********************************************************
 *
 *
 *  SOURCE FILE:    alphas.c
 *
 *
 *  FUNCTION:       int alphas(coef, filter_type,
 *                             filter_order, n)
 *
 *
 *  ARGUMENTS:
 *
 *      (input/output) COMPLEX coef[]
 *          Passes the approximating cosine coef-
 *      ficients and returns the filter impulse
 *      coefficients.
 *
 *      (input) int filter_type
 *          Type 1, 2, 3, or 4.
 *
 *      (input) int filter_order
 *          Number of filter impulse coefficients.
 *
 *      (input) int n
 *          The number of approximating cosines.
 *
 *
 *  RETURN:         int: OK
 *                       ERROR
 *
 *
 *  DESCRIPTION:
 *
 *      Converts the approximating cosine coefficients
 *      to the filter impulse coefficients based upon
 *      whether the filter is of type 1, 2, 3, or 4.
 *
 *      type 1: odd length / even symmetry
 *
 *      type 2: even length / even symmetry
 *
 *      type 3: odd length / odd symmetry
 *
 *      type 4: even length / odd symmetry
 *
 *
 *  DOCUMENTATION
 *  FILES:          None
 *
 *
 *  CONSTANTS:      OK    common.h
 *                  ERROR common.h
 *
 *
 *  MACROS
 *  EXPANDED:       None
 *
 *
 *  FUNCTIONS
 *  CALLED:         query()  screen.c
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:   17May88
 *
 *
 *  REVISIONS:      Ver 1.00
 *
 *
 **********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "screen.h"
#include "smath.h"


int alphas(COMPLEX coef[], int filter_type,
    int fil_order, int n)
{
    int i, j;

    double *work; /* temporary work space                  */

    if ((work = (double *) malloc(sizeof(double)
        * fil_order)) == NULL)
        {
        query("malloc error in alphas()");
        return(ERROR);
        }

    switch(filter_type)
        {
        case 1:
            for (i = 1; i <= n-1; ++i)
                {
                work[n-1-i] = 0.5 * coef[i].re;
            work[n-1+i] = work[n-1-i];
            }
        work[n-1] = coef[0].re;
        break;

    case 2:
        coef[0].re = coef[0].re + 0.5 * coef[1].re;
        for (i = 1; i <= n-2; ++i)
            coef[i].re = 0.5 * (coef[i].re +
                coef[i+1].re);
        coef[n-1].re = 0.5 * coef[n-1].re;
        for (i = 0; i <= n-1; ++i)
            {
            work[n-i-1] = 0.5 * coef[i].re;
            work[n+i] = work[n-i-1];
            }
        break;

    case 3:
        coef[0].re = coef[0].re - 0.5 * coef[2].re;
        for (i = 1; i <= n-3; ++i)
            coef[i].re = 0.5 * (coef[i].re -
                coef[i+2].re);
        coef[n-2].re = 0.5 * coef[n-2].re;
        coef[n-1].re = 0.5 * coef[n-1].re;
        for (i = 0; i <= n-1; ++i)
            {
            work[n-1-i] = 0.5 * coef[i].re;
            work[n+1+i] = -work[n-1-i];
            }
        work[n] = 0;
        break;

    case 4:
        coef[0].re = coef[0].re - 0.5 * coef[1].re;
        for (i = 1; i <= n-2; ++i)
            coef[i].re = 0.5 * (coef[i].re -
                coef[i+1].re);
        coef[n-1].re = 0.5 * coef[n-1].re;
        for (i = 0; i <= n-1; ++i)
            {
            work[n-1-i] = 0.5 * coef[i].re;
            work[n+i] = -work[n-1-i];
            }
        break;

    default:
        break;
    } /* end switch */
    for (i = 0; i <= fil_order-1; ++i)
        coef[i].re = work[i];

    free(work);
    return(OK);

}/* end alphas() */
```

```
/*************************************************************
 *
 *
 *  SOURCE FILE:    firres.c
 *
 *
 *  FUNCTION:       int fir_response()
 *
 *
 *  ARGUMENTS:      None
 *
 *
 *  RETURN:         int: OK
 *                       ERROR
 *
 *  DESCRIPTION:
 *
 *      Determines the FIR filters magnitude response
 *      by zero-padding the X register, which contains
 *      the impulse response coefficients, out to a
 *      power of two and then performs a forward, fast
 *      Fourier transform to find the frequency domain
 *      representation of the filter.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None
 *
 *
 *  CONSTANTS:      OK              common.h
 *                  ERROR           common.h
 *                  FIR_RESPONSE    rolf.h
 *                  CMPLX           smath.h
 *                  RECT            smath.h
 *
 *
 *  MACROS
 *  EXPANDED:       None
 *
 *
 *  FUNCTIONS
 *  CALLED:         get_reg()       stackops.c
 *                  clear_reg()     stackops.c
 *                  query()         screen.c
 *                  fft()           fft.c
 *                  check_pow2()    reg.c
 *
 *
 *  AUTHOR:         Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 *************************************************************/

#include <stdio.h>
#include "rolf.h"
#include "fir.h"

#define  FORWARD  1  /* forward  Fourier transform         */

/* template defined in fir.h, contains filter parameters  */
extern FIR_SPECS fir;

extern double freq_data[]; /* used when plottng response  */

/* reg_len is user adjustable, usually (1024, 2048, etc.) */
extern int reg_len;

int fir_response()
{
/*- template in rolf.h, x is a pointer to a ROLF register */
    REG *x;
    int i;

    x = get_reg(X); /* x now points to the X register     */

    x->len = reg_len;
    x->type = CMPLX; /* data will be complex after an fft */
    x->mode = RECT;  /* rectangular format by default     */
    x->contents = FIR_RESPONSE; /* index of a display msg */

/*------------------------------------------------------*/
/*  If the x register length is not a power of two, then  */
/*  set it to the next power of two with zero-padding. If */
/*  the register length cannot be made a power of two,    */
/*  clear x and abort.                                    */
/*------------------------------------------------------*/
    if (check_pow2(x))
        {
        clear_reg(x);
        return(ERROR);
        }

/*------------------------------------------------------*/
/*  Since the x register contains the filter coefficients,*/
/*  it must be at least as long as the filter order.      */
/*-------------------------------------------------------*/
    if (fir.order > x->len)
        {
        clear_reg(x);
        query("Error:default register length is too short");
        return(ERROR);
        }


    for (i = 0; i <= fir.order-1; ++i)
        x->reg[i].im = 0;

    for (i = fir.order; i <= x->len-1;++i)
        {
        x->reg[i].im = 0;
        x->reg[i].re = 0;
        }
    fft(x->reg, x->len, FORWARD);

/*- the second half of the response mirrors the first ----*/
    x->len /= 2;

    return(OK);

} /* end fir_response() */
```

```
/***********************************************************
 *
 *
 *  SOURCE FILE:   stackops.c
 *
 *
 *  FUNCTION:      None
 *
 *
 *  ARGUMENTS:     NA
 *
 *
 *  RETURN:        NA
 *
 *
 *  DESCRIPTION:
 *
 *      This source file contains a number of utilities
 *      which are germane to stack and register manipu-
 *      lations in the ROLF.EXE program.  Following is
 *      a terse listing of the functions:
 *
 *      int  create_regs(void)
 *      void free_regs(void)
 *      REG  *get_reg(int)
 *      void roll_up(void)
 *      void roll_down(void)
 *      void swap_reg(void)
 *      void copy_reg(REG *, REG *)
 *      void enter_reg(void)
 *      void disable_roll(void)
 *      void enable_roll(void)
 *      char *toggle_lock(void)
 *      void save_reg(void)
 *      void fetch_reg(void)
 *      void bin_stk_fix(void)
 *      void clear_stack(void)
 *      void clear_reg(REG *)
 *      void disp_stack(void)
 *
 *
 *  DOCUMENTATION
 *  FILES:         None
 *
 *
 *  CONSTANTS:     None
 *
 *
 *  MACROS
 *  EXPANDED:      None
 *
 *
 *  FUNCTIONS
 *  CALLED:        None
 *
 *
 *  AUTHOR:        Scott A. Nichols
 *
 *
 *  DATE CREATED:  17May88
 *
 *
 *  REVISIONS:     Ver 1.00
 *
 *
 ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "rolf.h"

#define  LOCKED   0  /* stack roll is disabled */
#define  UNLOCKED 1  /* stack roll is enabled  */

/*-- Define some external data. --------------------------*/
extern char *msg[];

static int stack_lock = UNLOCKED; /* let it roll initially*/

/*------------------------------------------------------*/
/*  This defines the variables and messages for         */
/*  displaying the register contents with disp_stack()  */
/*------------------------------------------------------*/
static char *locked =    "LOCKED",
             *unlocked = "UNLOCKED",
             *real =     "REAL",
             *cmplx =    "COMPLEX",
             *polar =    "POLAR",
             *rect =     "RECT",
             *null =     "",
             *mode,
             *type,
            *line = "------------------------------------"
                    "------------------------------------",
            *header = "¦ Register ¦   Length   ¦  Mode    ¦"
                        "  Type   ¦\t\tContents   \t ¦",
            *format = "¦    %-2s    ¦    %6d   ¦  %-7s ¦"
                        " %-7s ¦ %-25s ¦\n",
            *regs[] = {"X","Y","Z","T","R0"};
```

```
/*------------------------------------------------------------*/
/*  This defines the objects of which the programs            */
/*  registers consist of.  The complex pointer within         */
/*  such an object must still be allocated.                   */
/*------------------------------------------------------------*/
static REG x, y, z, t, r0, work;


/*============================================================*/
/*  Create the complex registers which constitute             */
/*  the stack.                                                */
/*============================================================*/
int create_regs()
{
    x.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (x.reg == NULL)
        {
        query("Memory not allocated !");
        return(ERROR);
        }

    y.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (y.reg == NULL)
        {
        query("Memory not allocated !");
        free(x.reg);
        return(ERROR);
        }

    z.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (z.reg == NULL)
        {
        query("Memory not allocated !");
        free(x.reg);
        free(y.reg);
        return(ERROR);
        }

    t.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (t.reg == NULL)
        {
        query("Memory not allocated !");
        free(x.reg);
        free(y.reg);
        free(z.reg);
        return(ERROR);
        }

    r0.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (r0.reg == NULL)
        {
        query("Memory not allocated !");
        free(x.reg);
        free(y.reg);
        free(z.reg);
        free(t.reg);
        return(ERROR);
        }

    work.reg = (COMPLEX *) malloc((unsigned)
        sizeof(COMPLEX) * REGLEN);
    if (work.reg == NULL)
        {
        query("Memory not allocated !");
        free(x.reg);
        free(y.reg);
        free(z.reg);
        free(t.reg);
        free(r0.reg);
        return(ERROR);
        }
    return(OK);
}


/*============================================================*/
/* Deallocate  the   stack.                                   */
/*============================================================*/
void free_regs()
{
    free(x.reg);
    free(y.reg);
    free(z.reg);
    free(t.reg);
    free(r0.reg);
    free(work.reg);
}


/*============================================================*/
/*  Make one of the stack registers available to a          */
/*  function.                                                 */
/*============================================================*/
REG *get_reg(int reg_choice)
{
    switch (reg_choice)
        {
        case X:
            return(&x);

        case Y:
            return(&y);

        case Z:
            return(&z);

        case T:
            return(&t);

        case R0:
            return(&r0);

        case WORK:
            return(&work);

        default:
            query("Invalid Register");
        }
}


/*===========================================================*/
/*  Roll the stack up one register.                          */
/*===========================================================*/
void roll_up()
{
    REG reg;

    if (stack_lock == LOCKED)
        return;
    reg = t;
    t = z;
    z = y;
    y = x;
    x = reg;
    return;
}


/*===========================================================*/
/*  Roll the stack down one register.                        */
/*===========================================================*/
void roll_down()
{
    REG reg;
    if (stack_lock == LOCKED)
        return;
    reg = x;
    x = y;
    y = z;
    z = t;
    t = reg;
    return;
}


/*===========================================================*/
/*  Swap the (X) and (Y) registers.                          */
/*===========================================================*/
void swap_reg()
{
    REG reg;

    reg = x;
    x = y;
    y = reg;
    return;
}



/*===========================================================*/
/*  Copy stack register utility.  Copies the contents of    */
/*  reg1 into reg2.                                          */
/*===========================================================*/
void copy_reg(REG *reg1, REG *reg2)
{
    int i;
    reg2->len = reg1->len;
    reg2->type = reg1->type;
    reg2->mode = reg1->mode;
    reg2->contents = reg1->contents;
    for (i = 0; i <= reg2->len-1; ++i)
        {
        reg2->reg[i].re = reg1->reg[i].re;
        reg2->reg[i].im = reg1->reg[i].im;
        }
    return;
}



/*===========================================================*/
/*  Perform an RPN  enter operation.                         */
```

```
/*==============================================================*/
void enter_reg()
{
    int i;
    if (stack_lock == LOCKED)
        return;
    roll_up();
    copy_reg(&y, &x);
    return;
}


/*==============================================================*/
/*  Disable stack roll.                                         */
/*==============================================================*/
void disable_roll()
{
    stack_lock = LOCKED;
}


/*==============================================================*/
/*  Enable stack roll.                                          */
/*==============================================================*/
void enable_roll()
{
    stack_lock = UNLOCKED;
}


/*==============================================================*/
/*  Toggle the lock which enables stack roll.                   */
/*==============================================================*/
char *toggle_lock()
{
     if (stack_lock == LOCKED)
         {
         stack_lock = UNLOCKED;
         return(unlocked);
         }
     else
         {
         stack_lock = LOCKED;
         return(locked);
         }
}


/*==============================================================*/
/*  Save the X register to R0.                                  */
```

```
/*==========================================================*/
void save_reg()
{
    int i;
    copy_reg(&x, &r0);
    return;
}



/*==========================================================*/
/*  Fetch the R0 register to X.                             */
/*==========================================================*/
void fetch_reg()
{
    int i;
    if (stack_lock == UNLOCKED)
        roll_up();
    copy_reg(&r0, &x);
    return;
}



/*==========================================================*/
/*  Binary stack fix up utility.                            */
/*==========================================================*/
void bin_stk_fix()
{
    REG temp;

    copy_reg(&t, &y);
    temp = y;
    y = z;
    z = t;
    t = temp;
}


/*==========================================================*/
/*  Clear the stack                                         */
/*==========================================================*/
void clear_stack()
{
    clear_reg(&x);
    clear_reg(&y);
    clear_reg(&z);
    clear_reg(&t);
    return;
}
```

```
/*==========================================================*/
/*  Clear a stack register.                                 */
/*==========================================================*/
void clear_reg(REG *reg)
{
    reg->len = 0;
    reg->type = REAL;
    reg->mode = RECT;
    reg->contents = NO_CONTENTS;
    return;
}


/*==========================================================*/
/*  Utility which displays the status of the stack          */
/*  registers.                                              */
/*==========================================================*/
void disp_stack()
{
    int i;
    printf("%s\n",line);
    printf("%s\n",header);
    printf("%s\n",line);
    if (r0.type == REAL)
        {
        type = real;
        mode = null;
        }
    else
        {
        type = cmplx;
        mode = (r0.mode == RECT ? rect : polar);
        }
    printf(format, regs[4], r0.len, mode, type,
        msg[r0.contents]);
    printf("%s\n",line);
    if (t.type == REAL)
        {
        type = real;
        mode = null;
        }
    else
        {
        type = cmplx;
        mode = (t.mode == RECT ? rect : polar);
        }
    printf(format, regs[3], t.len, mode, type,
        msg[t.contents]);
    if (z.type == REAL)
        {
        type = real;
        mode = null;
        }
    else
        {
        type = cmplx;
        mode = (z.mode == RECT ? rect : polar);
        }
    printf(format, regs[2], z.len, mode, type,
        msg[z.contents]);
    if (y.type == REAL)
        {
        type = real;
        mode = null;
        }
    else
        {
        type = cmplx;
        mode = (y.mode == RECT ? rect : polar);
        }
    printf(format, regs[1], y.len, mode, type,
        msg[y.contents]);
    if (x.type == REAL)
        {
        type = real;
        mode = null;
        }
    else
        {
        type = cmplx;
        mode = (x.mode == RECT ? rect : polar);
        }
    printf(format, regs[0], x.len, mode, type,
        msg[x.contents]);
    printf("%s\n",line);
    return;
}
```

A NEW METHOD FOR THE DESIGN OF FIR DIGITAL FILTERS

by

SCOTT ANTHONY NICHOLS

B.S. Kansas State University, 1986

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

ABSTRACT

Optimal FIR filter design procedures are time consuming to implement and compute. Standard frequency-sampling methods are efficient but lack control over transition band edges and often yield unsuitable characteristics at band-edges. This paper presents a procedure by which a suboptimal filter is obtained in one pass of the design algorithm. Control is maintained over the transition band edges and band-edge response behavior is smoothed by acknowledgment of ripple in pass-bands and stop-bands. This procedure makes use of Lagrange interpolation and an N/2-point DCT to obtain the filter coefficients. The method is faster than other known FIR design methods and often produces a usable filter when the frequency sampling method fails.